Concurrencia en la JVM*

Rubén Pinilla, Marisa Gil

Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya
c/ Jordi Girona, 1-3, Edifici D6 Campus Nord, 08034 Barcelona, Spain

e-mail: {rpinilla, marisa}@ac.upc.es

UPC-DAC-2003-5

30 de enero de 2003

ABSTRACT: La tecnología Java está siendo en estos años un punto de referencia importante para el desarrollo de aplicaciones. La gran aceptación de Java por la comunidad de desarrolladores de software se basa, principalmente, en sus características como entorno de ejecución independiente de la plataforma. Entre ellas se encuentra el soporte, a nivel lenguaje, de concurrencia que permite ejecutar aplicaciones multithreaded. En este report presentamos los elementos que ofrece Java para construir programas concurrentes, las abstracciones realizadas por la JVM y el análisis de la implementación de la JVM de Sun (Java 2 SDK 1.2.2-006). A partir de este estudio y de un ejemplo en varias plataformas, mostraremos que el API que ofrece la JVM es independiente desde el punto de vista del programador pero con un comportamiento distinto, dependiendo de la plataforma en la que se ejecute. Esta situación puede llevar a un mal funcionamiento de los programas, que se solucionaría implementando un HPI que fuera capaz de añadir un grado de abstracción con respecto a la plataforma de ejecución, de manera que usando un mismo HPI para diferentes plataformas conseguiríamos minimizar los efectos de dependencia observados en este trabajo, consiguiendo así una política de gestión uniforme de las entidades de planificación ofrecidas por los distintos sistemas operativos.

KEYWORDS: Multithreaded, Java Virtual Machine, user threads, kernel threads, scheduling, concurrency, Java threads, Java monitor.

* This work has been supported by the Ministry of Science and Technology of Spain and by the European Union (FEDER) under contract TIC2001-0995-C02-01. The Itanium based computers are a HP/Intel grant to the Computer Architecture Department.

Índice

Índice de figuras

Índice de tablas

1. Introducción

Java fue creado en los laboratorios de Sun Microsystems Inc. para intentar contrarrestar la incompatibilidad entre plataformas con las que se encontraban los desarrolladores de software. Existen varias versiones sobre el origen de Java [5], pero quizás la más difundida es la que hace referencia al interés que tuvo Sun Microsystems por el mundo de la electrónica de consumo y al desarrollo de programas para dispositivos electrónicos.

Normalmente el término **Java** siempre se asocia al lenguaje de programación, pero Java no es sólo un lenguaje de programación. Java es una arquitectura formada por un conjunto de cuatro tecnologías interrelacionadas entre sí que se presenta en la siguiente ilustración.

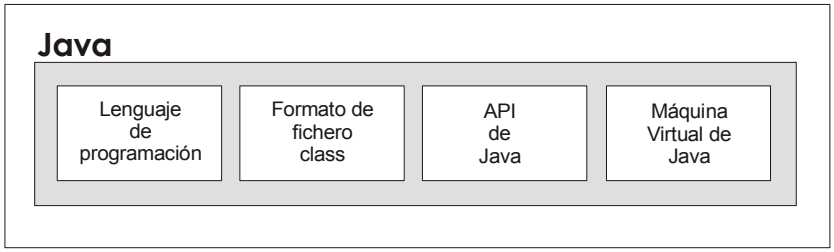

Figura 1-1. Arquitectura de Java formada por cuatro tecnologías interrelacionadas entre sí.

El proceso de desarrollo de un programa en Java pasa, principalmente, por dos etapas (Compilación de Java y Ejecución de Java) que diferencian dos entornos de trabajo:

Java en tiempo de compilación: Una vez tenemos el programa escrito en lenguaje Java, codificado en ficheros con extensión `.java`, se procede a compilarlo mediante el compilador de Java. El resultado de la compilación son los ficheros objeto con extensión `.class` que contienen la traducción de Java a **bytecode** (lenguaje que entiende la Máquina Virtual de Java).

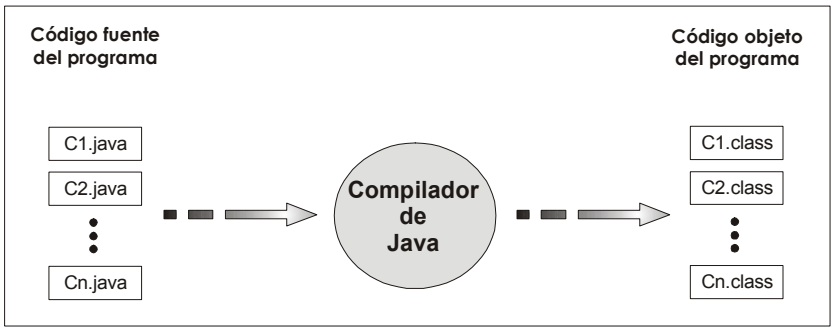

Figura 1-2. Entorno de Java en tiempo de compilación. Compilación de una aplicación formada por n archivos de código fuente, dando como resultado n archivos de código objeto.

Java en tiempo de ejecución: Después de la compilación del programa, disponemos de los ficheros `.class` que, junto con los ficheros `.class` que forman el API[1] de Java, serán ejecutados

[1] De Application Programing Interface.

por la **Máquina Virtual de Java**[2] (máquina abstracta que ejecuta instrucciones codificadas en bytecode). La **Plataforma Java**, formada por el API de Java junto con la JVM, debe estar presente en cualquier ordenador o dispositivo que quiera ejecutar programas desarrollados en Java.

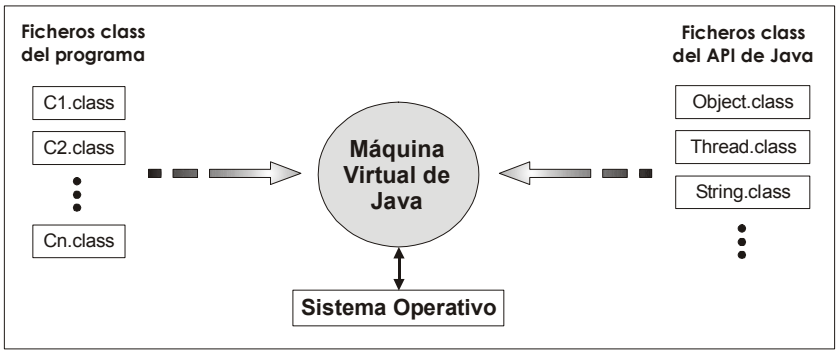

Figura 1-3. Entorno de Java en tiempo de ejecución. Ejecución de una aplicación, formada por n archivos de código objeto (.class), por la Plataforma Java formada por la JVM y el API de Java.

Una vez vistos los entornos de Java, vamos a describir las cualidades de Java como lenguaje de programación que nos podrán dar una visión global de sus posibles aplicaciones en el mundo del software:

- **Simple**: Elimina características de otros lenguajes como C o C++ tales como la aritmética de punteros, las referencias, los tipos estructurados (tuplas), definición de tipos (`typedef` de C), definición de macros (los `#define` de C) y la liberación de memoria dinámica (`free`).

- **Orientado a objeto**: Implementa la tecnología básica de C++, soportando encapsulación, herencia y polimorfismo. Como en C++, las plantillas de objetos son llamadas clases y sus copias instancias, que necesitan ser creadas y destruidas en espacio de memoria.

- **Distribuido**: Posibilita la creación de aplicaciones distribuidas, mediante el uso de librerías que el lenguaje proporciona, capaces de ejecutarse en varias máquinas conectadas a través de una red.

- **Robusto**: Realiza comprobaciones de código tanto en tiempo de compilación como de ejecución. La declaración de los tipos de los métodos es explícita, reduciendo así la posibilidad de error. Desaparece la opción de gestionar la memoria por parte del programador, eliminando así el posible acceso a direcciones de memoria inválidas y una mala gestión de memoria.

- **Independiente de la arquitectura**: El compilador de Java genera código objeto (bytecode) independiente de la plataforma en la que se vaya a ejecutar. Será, en última instancia, el intérprete de Java implementado para cada plataforma (también llamado run-time de Java o JRE) quien ejecute el programa. La implementación del interprete se basa principalmente en la creación de una Máquina Virtual de Java que proporcionará

[2] Del inglés Java Virtual Machine. A partir de ahora se referenciará mediante JVM.

una abstracción y encapsulación de las funcionalidades que ofrezca la plataforma de ejecución a través del sistema operativo.

- **Seguro**: Inherente al lenguaje, ya que elimina características como los punteros y el casting implícito de tipos. Por otra parte, el código Java a la hora de ejecutarse pasa por diversas comprobaciones por parte del Verificador de bytecode (examina la posible existencia de código ilegal) y por el Cargador de Clases (se separan los recursos locales y los remotos) que mantiene en espacios de nombres privados (asociados al origen) para cada clase importada desde la red y mantiene un orden de búsqueda de clases local-remoto (cuando una clase accede a otra clase, primero busca en las clases predefinidas y luego en el espacio de nombres de la clase que hace referencia).

- **Portable**: Además de la independencia de la arquitectura, cosa que posibilita en gran medida la portabilidad, Java implementa otros estándares que hacen más fácil la construcción de software portable como, por ejemplo: los enteros son de 32 bits en complemento a 2 y el sistema abstracto de ventanas posibilita su implantación en entornos como Unix, PC o Mac.

- **Interpretado**: Se trata de un lenguaje interpretado, de forma que la compilación no genera un código objeto directamente ejecutable por el sistema operativo. Para ello es necesaria la existencia de un intérprete que lo ejecute.

- **Multitarea**: Java posibilita, a nivel de lenguaje, implementar aplicaciones que tengan varios flujos (threads) de ejecución que se ejecuten concurrentemente. Para ello, el lenguaje proporciona una serie de elementos para poder trabajar con flujos y mecanismos de sincronización.

- **Dinámico**: Java mantiene independientes los módulos que forman parte de una aplicación hasta el momento de ejecutar dicha aplicación. De esta forma, nuevas versiones de librerías harán que aplicaciones desarrolladas anteriormente puedan seguir siendo ejecutadas (siempre y cuando estas librerías mantengan el API anterior).

El lenguaje Java puede pensarse como un lenguaje de propósito general que está orientado para trabajar en red. De tal forma, Java posee algunas características que pueden descartarlo frente a otros lenguajes a la hora de programar una aplicación. A continuación comentaremos algunos de sus inconvenientes:

- La interpretación de bytecode produce una ejecución más lenta que la ejecución de la misma aplicación desarrollada en un lenguaje compilado. Con la aparición de la técnica **JIT**[3] se acelera la interpretación de bytecode.

- Los programas en Java se enlazan dinámicamente, existiendo así la posibilidad de que la JVM tenga que descargar ficheros .class remotos aumentando el tiempo de ejecución de los programas.

- Las comprobaciones tales como los límites de los arrays y las referencias a objetos se realizan en tiempo de ejecución. Las comprobaciones en tiempo de ejecución no pueden eliminarse (mayor tiempo de ejecución).

[3] Del inglés Just In Time.

- La memoria dinámica se gestiona a través del mecanismo de garbage collector. En lugar de ser el programador quien, mediante sentencias del estilo `Objeto.free()`, libere explícitamente la memoria ocupada por los objetos, se deja dicha tarea a un thread de la JVM dedicado a realizar la tarea de garbage collector. La necesidad de tiempo de CPU por parte de este thread para poder acometer su función añade cierto grado de incertidumbre a la ejecución de un programa multiflujo.

2. Estructura interna de la JVM

La especificación de la Máquina Virtual de Java [6] define el comportamiento de la misma mediante la descripción de subsistemas, áreas de datos en tiempo de ejecución, interfaces e interacción entre ellos. La siguiente ilustración muestra la estructura interna de la JVM:

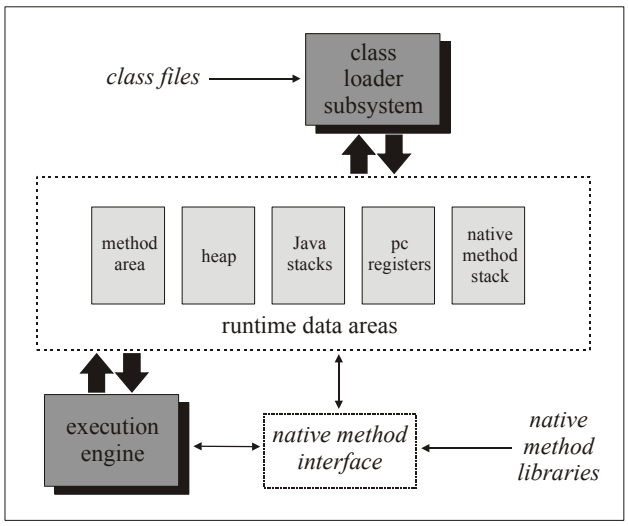

Figura 2-1. Estructura de la JVM formada por subsistemas, áreas de datos en tiempo de ejecución, interfaces y la interacción entre ellos. (Bill Venders [2]).

Class Loader Subsystem (Subsistema cargador de clases): Un mecanismo para poder cargar clases e interfaces a través de referencias a nombres completos.

Execution Engine (Mecanismo de ejecución): Su misión es la de ejecutar las instrucciones que contienen los métodos de las clases que han sido cargadas por parte del Class Loader Subsystem.

Runtime Data Areas (Áreas de datos en tiempo de ejecución): es la forma que tiene de organizar la memoria la JVM.

Una instancia de la JVM tiene una **Method Area** y un **Heap**. Estas zonas de memoria son compartidas por todos los threads que se ejecutan en la JVM. Cuando la máquina virtual carga un fichero class, extrae información sobre el tipo y la almacena en la Method Area. A medida que la ejecución del programa evoluciona, la JVM carga todas las instancias de los objetos dentro del Heap.

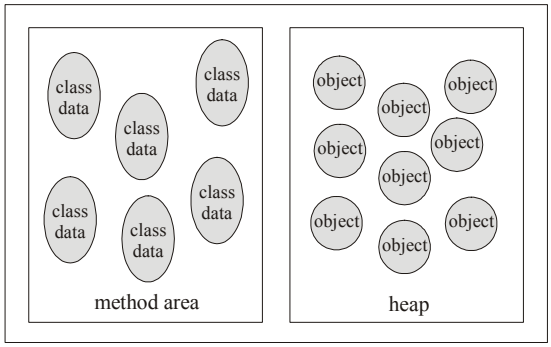

Figura 2-2. Zonas de memoria (method area y heap) de acceso compartido por todos los threads. (Bill Venders [2]).

Cada thread nuevo que se cree, dispondrá de su propio **pc register** (registro para el contador de programa) y de una **Java stack** (pila de Java). Cuando un thread esté ejecutando un método no nativo, el valor del pc contendrá una referencia a la siguiente instrucción a ejecutar. La Java stack almacenará el estado de las invocaciones de métodos no nativos de Java que el thread vaya haciendo durante su ejecución. Este estado incluye a las variables locales, los parámetros de los métodos, el valor de retorno (en caso de haberlo) y cálculos intermedios. El estado de invocación de métodos nativos se guarda usando unas estructuras de datos llamadas **native method stacks**.

La Java stack está formada por **stack frames** (también llamadas **frames**). Una stack frame guarda el estado de una invocación de un método. Cuando un thread invoca a un método, la JVM empila una nueva stack frame en la Java stack de ese thread y, cuando finalice la ejecución de dicho método, la JVM desempilará dicha stack frame de la Java stack.

La JVM no dispone de registros para almacenar cálculos intermedios, con lo que el conjunto de instrucciones de la JVM usa la Java stack para almacenar esos posibles cálculos intermedios que genere la ejecución de cada thread.

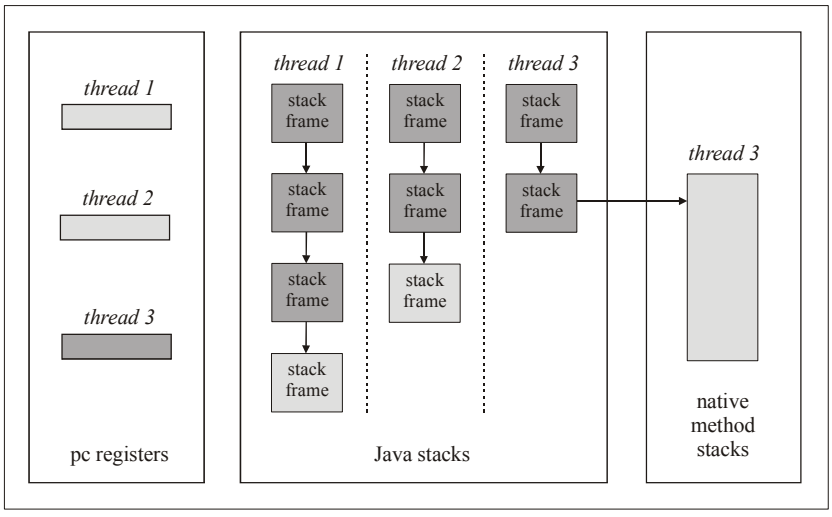

Figura 2-3. Zonas de memoria (pc registers, Java stacks y native method stacks) de acceso exclusivo para cada thread. (Bill Venders [2]).

Una instancia de la JVM empieza a ejecutar la aplicación Java invocando el método `main`. Cualquier clase que tenga definido el método `main`, declarado de la forma:

```
public static void main(String[] args)
```

puede ser usada como punto de comienzo de una aplicación Java.

El método `main` estará asignado al thread inicial de la aplicación, que tendrá entre otras capacidades la de iniciar el resto de threads de la aplicación.

Hay dos tipos de threads dentro de la máquina virtual: **daemon** y **non-daemon**. Un daemon thread normalmente es usado por la propia maquina virtual, realizando las funciones de **garbage collection** (proceso automático que se encarga de liberar el espacio ocupado por los objetos que ya no son referenciados dentro de la aplicación Java). De todas formas, la aplicación Java puede marcar un thread como daemon (el thread inicial de una aplicación es un thread del tipo non-daemon).

La instancia de la máquina virtual continúa activa mientras haya algún thread non-daemon que se esté ejecutando. Cuando todos los threads non-daemon de la aplicación terminen, entonces la instancia de la máquina virtual de java finalizará su ejecución.

En el siguiente punto vamos a ver los elementos que, en el ámbito de lenguaje de programación, nos ofrece Java para construir aplicaciones concurrentes y ver la forma que tiene para manejar flujos de ejecución y mecanismos de sincronización.

3. Concurrencia en Java

Como se ha dicho anteriormente, una de las características destacable del lenguaje Java es el soporte de elementos para la programación multithreaded (multiflujo) a nivel del lenguaje. Esta característica no es novedosa, ya que lenguajes como Ada ya ofrecían la posibilidad de describir flujos de ejecución a través del lenguaje. A continuación vamos a describir dichos elementos dando algunas definiciones y características para su identificación.

Empezaremos por la definición de **thread**, que podemos definirlo como un flujo de control independiente dentro de un proceso, formado por un contexto (un conjunto de registros, una pila y un pc) y una secuencia de instrucciones que se puede ejecutar independientemente de otras secuencias de instrucciones. La estructura de datos que agrupa las estructuras de datos que tiene un thread se suele llamar **TCB** (Thread Control Block).

Un **proceso** es la entidad de asignación de recursos del sistema que, en un entorno multithreaded, es considerado como un marco de ejecución. El proceso dispone de un espacio de direcciones y otros recursos de sistema tales como una tabla de descriptores de ficheros abiertos, tabla de rutinas de atención a signals, etc. Cuando un proceso es creado en un sistema operativo multiflujo, inicialmente sólo existe un thread que sería el equivalente al modelo de proceso de un sistema operativo no multiflujo. Cada proceso se gestiona a través de una estructura de datos llamada **PCB** (Process Control Block), que aglutina las estructuras de datos y referencias a otras estructuras que forman al proceso.

Los threads comparten los recursos asignados al proceso que los contiene (espacio de direcciones y zona de código). Por otra parte, cada thread dispone de una zona de memoria privada, atributos de planificación y una pila a los cuales no pueden acceder el resto de threads.

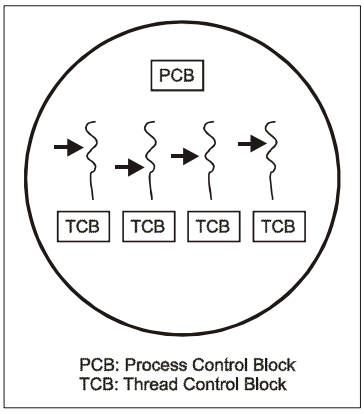

Figura 3-1. Proceso gestionado a través de su PCB (Process Control Block) formado por 4 threads que comparten acceso al PCB del proceso y son gestionados cada uno con un TCB (Thread Control Block), que es privado para cada thread.

Cabe hacer distinción entre threads de usuario (user threads) y threads de sistema (kernel threads):

- **Thread de usuario**: Son gestionados a través del API de una librería de threads, existen sólo en espacio de usuario y no son visibles por el sistema operativo. Para que un thread de usuario pueda ejecutarse en un procesador debe estar asociado a una entidad planificada por el sistema (kernel-scheduled entity). Esta entidad planificada por el sistema puede ser un proceso con un thread (modelo tradicional de proceso) o un thread de sistema. En algunos sistemas a los threads de usuario se les llama simplemente threads.

- **Thread de sistema**: Es una entidad planificada y gestionada por el scheduler del sistema. Son creados también a través del API de una librería de threads, pero al ser entidades planificadas por el sistema son visibles y planificados por el sistema operativo. También a los threads de sistema se les suele llamar **LWP** (**LightWeight Process**, procesos ligeros).

Definimos **modelo de threads** como la forma de mapear threads de usuario sobre threads de sistema. Hay tres modelos de threads que describen las diferentes formas de relacionar threads de usuario con threads de sistema:

- **M a 1** : Varios threads de usuario a un thread de sistema.

- **1 a 1**: Un thread de usuario a un thread de sistema.

- **M a N**: Varios threads de usuario a varios threads de sistema.

El mapeo entre threads de usuario y threads de sistema se realiza usando **procesadores virtuales**. Un procesador virtual (PV) es una entidad de librería que normalmente es implícita. Para un thread de usuario, un PV se comporta como una CPU para un thread de sistema. Así, un PV en la

librería de threads es un thread de sistema o una estructura de datos asociada con un thread de sistema.

La forma como se mapea un thread de usuario sobre un thread de sistema se llama **contention scope**. Existen dos formas posibles de contention scope:

- **System contention scope** (o global contention scope): Un thread de usuario se mapea directamente sobre un thread de sistema. Todos los threads de usuario del modelo 1 a 1 usan este tipo de mapeo.

- **Process contention scope** (o local contention scope): Un thread de sistema es compartido por varios threads de usuario. Este es el mapeo usado por los threads de usuario del modelo M a 1.

Modelo M a 1

Las implementaciones de este modelo de threads permiten a la aplicación crear varios threads en espacio de usuario que se ejecutan concurrentemente. Los threads, al crearse en espacio de usuario, tienen la limitación de que tan sólo un thread puede acceder al sistema operativo en un momento dado. De esta forma, desde el punto de vista del sistema operativo, sólo existe una entidad planificable por el sistema operativo.

Este modelo permite la ejecución concurrente de M threads creados en espacio de usuario, pero al estar limitados por 1 entidad planificable por el SO no hay posibilidad de ejecución paralela en un sistema multiprocesador.

Algunas ventajas de este modelo:

- **Sincronización**: Cuando un thread de usuario desea realizar una sincronización, la librería de threads de usuario comprueba si dicho thread necesita bloquearse. Tanto si el thread debe o no bloquearse, la librería no realizará llamadas al sistema operativo para poder llevar a cabo el bloqueo.

- **Creación de threads**: En la creación de un thread, la librería tan sólo debe crear un contexto (una pila y registros) para el thread y encolarlo en la cola de threads disponibles.

- **Eficiencia en el uso de recursos**: La memoria de espacio de sistema no se usa para crear las pilas de los threads de usuario. Con lo cual, esto permite la creación de tantos threads de usuario como pueda albergar la memoria virtual del sistema.

- **Portabilidad**: Las librerías de threads de usuario se pueden implementar haciendo llamadas a librerías estándar que estén presentes en gran número de sistemas.

Algunas desventajas de este modelo:

- **Interfaz monoflujo con el sistema operativo**: Debido a que sólo hay un thread de sistema, si un thread de usuario realiza una llamada bloqueante al sistema

operativo, el proceso entero se bloquerá impidiendo así la posibilidad de hacer progreso en su ejecución al resto de threads de usuario.

- **No hay paralelismo**: aplicaciones con varios flujos de ejecución que usen este modelo de threads no se ejecutarán más rápido en sistemas multiprocesador, debido a que el thread de sistema es insuficiente para el uso óptimo de los procesadores del sistema.

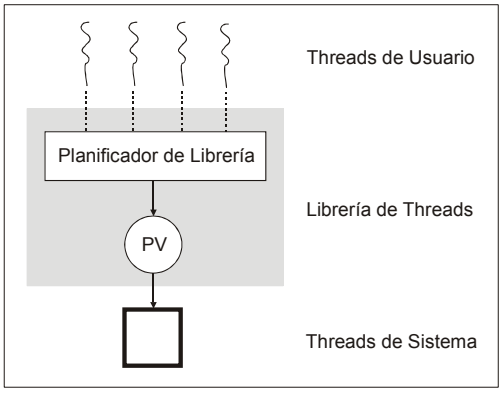

Figura 3-2. Modelo de threads M a 1. Existen M threads de usuario que se planifican sobre 1 PV (Procesador Virtual), relacionado el PV con 1 thread de sistema.

Modelo 1 a 1

En este modelo, cada thread creado por la aplicación es conocido por el sistema operativo como una entidad planificable para ser ejecutada. Así, todos los threads pueden acceder al mismo tiempo al sistema operativo, careciendo de la limitación del modelo M a 1 en cuanto a la posibilidad de la ejecución paralela en un sistema multiprocesador.

Como inconveniente principal se encuentra el hecho de que la creación de un thread es costosa debido a que por cada nuevo thread que se crea en espacio de usuario se debe crear una nueva entidad planificable por parte del sistema operativo y esto resulta costoso. Normalmente, los paquetes de threads que implementan este modelo suelen limitar el número máximo de threads que pueden soportarse por el sistema.

Algunas ventajas de este modelo:

- **Paralelismo escalable**: Como cada thread de sistema es una entidad planificada por el sistema, varios threads pueden ejecutarse concurrentemente en diferentes procesadores. Con lo cual, cuando una aplicación use una librería de threads (modelo de threads 1 a 1) puede conseguir notable mejora cuando se migra desde un sistema monoprocesador a un sistema multiprocesador.

- **Interfaz multiflujo con el sistema operativo**: A diferencia del modelo M a 1, cuando un thread de usuario y su thread de sistema se bloquea, no impide progresar en la ejecución al resto de threads.

Algunas desventajas de este modelo:

- **Sincronización costosa**: Debido a que los threads de sistema necesitan que el sistema operativo gestione su planificación, la sincronización entre threads de sistema requiere hacer una llamada al sistema para bloquear dicho thread si no se adquiere el lock inmediatamente.

- **Creación costosa**: La creación de un thread de usuario requiere la creación de un thread de sistema, consumiendo así recursos del sistema.

- **Ineficiencia en el uso de recursos**: Cada thread creado por la aplicación requiere memoria de sistema para su pila y demás estructuras de datos asociadas con el thread de sistema.

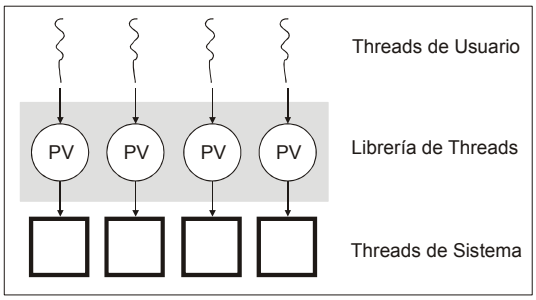

Figura 3-3. Modelo de threads 1 a 1. Cada thread de usuario que se vincula a 1 PV (Procesador Virtual), relacionado cada PV con 1 thread de sistema.

Modelo M a N

Este modelo, también llamado modelo de 2 niveles, mitiga los costes de creación de los threads creados en espacio de usuario. A su vez, también se reduce el tamaño de los threads reduciendo así el tamaño del proceso que los contiene.

Las librerías de threads que implementan este modelo suelen tener un mecanismo de scheduling de threads de usuario por encima del sistema operativo. Con lo cual, el sistema operativo sólo necesita gestionar los threads que estén activos. Este modelo disminuye el coste de programación de aplicaciones multithreading debido a que el programador no tiene restringido a usar un número limitado de threads para su aplicación.

En definitiva, este modelo intenta aprovechar las ventajas de los modelos M a 1 y 1 a 1. Por el contrario, intenta minimizar las desventajas de estos dos modelos.

En este modelo de threads, aparece el concepto de **nivel de concurrencia** definiendo el número de procesadores virtuales que se utilizan para ejecutar threads de usuario con process contention scope. El nivel de concurrencia no puede superar al número de threads de usuario con process contention scope y suele ser establecido dinámicamente por la librería de threads.

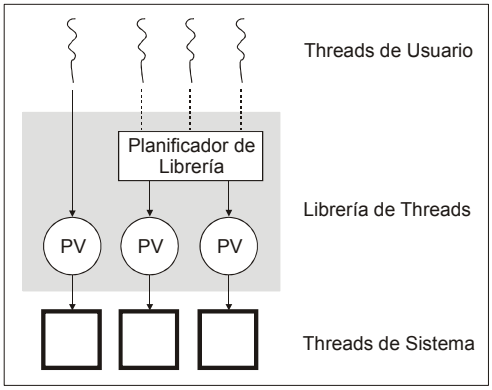

Figura 3-4. Modelo de threads M a N. En la ilustración se muestran 3 threads de usuario planificados sobre 2 Procesadores Virtuales (PV) y 1 thread de usuario relacionado con 1 PV. Cada PV está vinculado con 1 thread de sistema.

Focalizando nuestra atención en la implementación de la JVM, vemos que en el momento de su compilación se elige un modelo de threads mediante el cual gestionará todos los threads. Existen 2 alternativas de modelos de threads posibles entre los cuales se debe elegir uno. Los modelos de threads son los siguientes:

- **Green threads** (GT): Modelo de threads a nivel de usuario.
- **Native threads** (NT): Modelos de threads dependiente de los threads ofrecidos por el sistema operativo.

Estos modelos de threads, tienen la siguiente correspondencia con los modelos de threads anteriormente vistos:

- **Green threads** (GT): Modelo M a 1.
- **Native threads** (NT): Modelos 1 a 1 y M a N (depende del sistema operativo).

Sistema Operativo	**Modelos de Threads**		
	GT	NT	
	M a 1	1 a 1	M a N
Linux	✔	✔	
Solaris	✔		✔
Win32		✔	

Tabla 3-1. Modelos de threads soportados por varios sistemas operativos, haciendo referencia a la posible implementación del modelo de threads de la JVM (GT: Green Threads y NT: Native Threads).

En el caso de Java, dentro del API de Java se define la clase Thread (dentro del paquete java.lang) mediante la cual es posible definir flujos de ejecución. La descripción detallada de esta clase se encuentra en el Apéndice A. En la tabla que se muestra a continuación se encuentran las principales características de los threads de Java:

Categoría	Descripción

Clase Thread	- Métodos de control:
	  - Crear un thread (constructor `Thread`).
	  - Iniciar la ejecución (`start`).
	  - Detener la ejecución (`suspend`, `interrupt`, `stop`, `sleep`).
	  - Reanudar la ejecución (`resume`).
	  - Ceder el procesador (`yield`).
	  - Esperar la terminación de otro thread (`join`).
	- Métodos de consulta y modificación de propiedades:
	  - Nombre (`getName` / `setName`).
	  - Prioridad (`getPriority` / `setPriority`):
	  ▪ Prioridad mínima: `MIN_PRIORITY` (`1`).
	  ▪ Prioridad normal: `NORM_PRIORITY` (`5`), la asignada por defecto.
	  ▪ Prioridad máxima: `MAX_PRIORITY` (`10`).
	  - ThreadGroup (`getThreadGroup`): Cada thread pertenece a un ThreadGroup.
	  - Daemon (`isDaemon` / `setDaemon`): Indica si el thread es daemon o no.

Planificación	- Política de planificación: Por prioridad y con preemción. - La prioridad es estática: El algoritmo de planificación no modifica la prioridad de los threads. - Se aplica round robin para 2 threads con igual prioridad.
Sincronización	- La sincronización entre threads se ofrece a través de los métodos `wait`, `notify` y `notifyAll`, heredados de la clase `java.lang.Object`.

Tabla 3-2. Características principales de los threads ofrecidos por Java.

Para llevar a cabo la compartición de datos entre threads, es necesaria la entrada en juego de elementos que nos proporcionen un aislamiento que eviten efectos colaterales provocados por la interferencia entre los threads que intervengan durante el proceso de cálculo.

Así, definimos **sincronización** como un conjunto de actividades de coordinación y acceso a datos compartidos entre múltiples threads. Podemos hacer distinción entre dos tipos de sincronización:

- **Exclusión mutua**: Permite la ejecución concurrente de varios threads compartiendo datos de forma que se ejecuten sin interferir en el trabajo de cada thread. Es decir, que la ejecución del proceso de cálculo de forma secuencial dé el mismo resultado que la ejecución en su versión concurrente. Este tipo se sincronización es soportado a nivel de lenguaje Java por el uso de **synchronized methods** y **synchronized blocks**, que a nivel de la JVM requiere el uso de **locks**.

- **Cooperación**: Habilita la ejecución de un conjunto de threads de forma conjunta hacia un mismo objetivo. La cooperación es ofrecida, a nivel de lenguaje Java, a través del uso de los métodos `wait`, `notify` y `notifyAll` de la clase `java.lang.Object`. A nivel de la JVM, dichos métodos requieren el uso de **wait sets**.

Tipo de sincronización	¿Cómo se implementa?		
	Lenguaje Java	Máquina Virtual de Java	
Exclusión mutua	synchronized method synchronized block	monitor	lock
Cooperación	Métodos `wait`, `notify` y `notifyAll` de la clase `java.lang.Object`		wait set

Tabla 3-3. Tipos de sincronizaciones ofrecidas por Java y cómo las implementa.

En el caso de Java, la sincronización viene ofrecida mediante una abstracción llamada **monitor**[4], formado por un **lock** y un **wait set** [4] y que está presente en todas las instancias de la clase `java.lang.Object` (y las clases derivadas de ésta). Una definición más gráfica [2] de monitor de Java es la que se define como un edificio que contiene una habitación especial (normalmente contiene datos) que puede ser ocupada por sólo un thread en un momento dado. Desde el momento en que un thread entra en esta habitación hasta que la abandona, el thread tiene acceso exclusivo sobre cualquier dato que contenga la habitación. Un thread dentro de un monitor puede realizar las siguientes acciones, que son mostradas gráficamente en la **Figura 3-5:**

Acción	Descripción
Enter	El thread compite con otros threads para adquirir el lock.
Acquire	El thread adquiere el lock.
Own	El thread ejecuta el código asociado a la región del monitor.
Release	El thread libera el lock.
Exit	El thread abandona el monitor.

Tabla 3-4. Acciones que puede realizar un thread sobre un monitor.

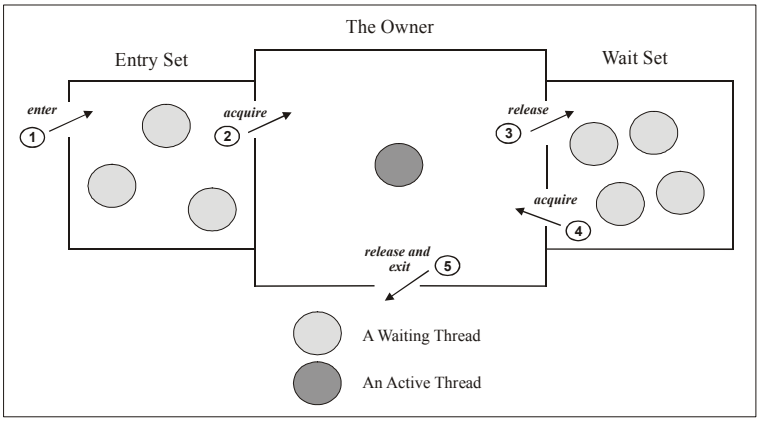

Figura 3-5. Estructura de un monitor de Java, constituido por un lock (Entry Set) y un wait set. (Bill Venders [2]).

Un monitor, además de estar asociado a un conjunto de datos, también está asociado a una zona de código que, en la obra [3] se califica como **monitor region** (región del monitor). Una región

[4] El concepto monitor fue desarrollado en los años 70 principalmente por Per Brinch Hansen, C. A. R. Hoarse y E. W. Dijkstra [7]. El mecanismo de sincronización en Java difiere del definido por Brinch-Hansen y Hoarse. En [8] se describe una posible implementación de monitor en lenguaje Java a semejanza de cómo fue definido por dichos autores.

del monitor es un conjunto de instrucciones que necesita ser ejecutado como una operación indivisible con respeto a un monitor en concreto. Es decir, un thread ha de ser capaz de ejecutar la región del monitor desde el principio hasta el final de tal forma que ningún otro thread pueda ejecutar concurrentemente la región del mismo monitor. La única forma que tiene un thread de ejecutar la región de un monitor es adquirir el monitor.

Cuando un thread llega al principio de la región del monitor se sitúa en un **entry set** (conjunto de entrada) asociado al monitor. Un entry set se podría definir, haciendo alusión a la definición de monitor, como el vestíbulo del edificio (concretamente como un lock). Si ningún otro thread está esperando en el entry set y ningún thread posee el monitor, el thread adquiere el monitor y continúa ejecutando la región del monitor. Cuando el thread finaliza la ejecución de la región del monitor, lo libera y sale del monitor.

Si un thread llega al principio de la región del monitor que está protegida por un monitor que ya posee otro thread, el nuevo thread debe esperar en el entry set. Cuando el thread que actualmente posee el monitor salga del monitor, el nuevo thread llegado debe competir con el resto de threads que están esperando en el entry set para adquirir el monitor (sólo un thread podrá adquirirlo).

La acción de adquirir o liberar un lock viene controlada, en el lenguaje de programación Java, mediante el uso de la palabra clave `synchronized`. Tenemos dos construcciones sintácticas para utilizar la palabra `synchronized`:

- **Synchronized methods**:

```
synchronized void incrementar()
{
     a++;
}
```

- **Synchronized blocks**:

```
void incrementar()
{
     synchronized(this)
     {
          a++;
     }
}
```

En cambio, el wait set del monitor se manipula mediante el uso de los métodos `wait`, `notify` y `notifyAll`. Estos métodos sólo pueden ser invocados una vez se haya adquirido el lock del objeto referenciado, es decir, sólo pueden llamarse dentro de un synchronized method o un synchronized block.

La forma de monitor usada por la Máquina Virtual de Java se llama "**Wait and Notify**" **monitor**. En ocasiones, el "Wait and Notify" monitor también es llamado "**Signal and Continue**" **monitor** debido a que después de que un thread llama a notify (signal), retiene la propiedad del monitor y continúa ejecutando la región del monitor (continue). Un tiempo después, el thread notificador libera el monitor y un thread que estaba esperando es despertado.

En este tipo de monitores, un thread que actualmente posee el monitor puede suspender su ejecución dentro del monitor ejecutando el método `wait`. Cuando un thread ejecuta dicho método,

libera el monitor y entra en el **wait set**. El thread permanecerá suspendido en el wait set hasta que otro thread ejecute `notify` dentro del monitor. Cuando un thread llama a `notify`, continúa poseyendo el monitor hasta que lo libere bien ejecutando un `wait` o finalizando la ejecución de la región del monitor. Después de que el thread que ejecutó `notify` haya liberado el monitor, el thread que estaba esperando será despertado y readquirirá el monitor.

Método	Descripción
`wait`	- Si el thread actual ha sido interrumpido, entonces el método retorna lanzando una excepción InterruptedException. En otro caso, el thread actual se bloquea. - La JVM añade el thread bloqueado al entry set del objeto referenciado. - Se libera el lock asociado al objeto referenciado.
`notify`	- Si existe algún thread esperando en el wait set del objeto referenciado, la JVM saca a un thread. - El thread sacado del wait set debe adquirir de nuevo el lock del objeto referenciado. - Se restaura al thread en el punto en que hizo `wait` y continúa su ejecución.
`notifyAll`	- La JVM saca a todos los threads del wait set. - Cada thread sacado del wait set debe adquirir de nuevo el lock del objeto referenciado, compitiendo con los demás. - Se restaura al thread que adquiere el lock en el punto en que hizo `wait`.

Tabla 3-5. Descripción de la implementación de los métodos de la clase `java.lang.Object` utilizados para sincronizar la ejecución entre threads.

Una llamada a `wait`, por parte de un thread (sea T1 el thread), pasa por las siguientes acciones (ver **Figura 3-6**):

- Cuando T1 llega al inicio del synchronized block compite con los demás threads para adquirir el lock (**1. enter**).
- Una vez ha adquirido el lock (**2. acquire**), T1 pasa a poseer el monitor (**3. own**).
- T1 empieza a ejecutar la región del monitor y llega al punto de la llamada a `wait`. La ejecución de `wait` hace que T1 suspenda su ejecución, se añada al wait set del monitor y libere el lock (**4. release**).
- En este punto, T1 permanecerá bloquetado hasta que otro thread ejecute `notify` sobre el objeto referenciado por T1. Sea T2 este thread, con lo que T1 continúa su ejecución.
- T1 debe readquirir el lock (**5. acquire**) y, una vez tenga el lock, continuará con la ejecución justo después de la llamada a `wait` (**6. own**).
- Finalmente, T1 libera al lock (**7. release**) y sale del monitor (**8. exit**).

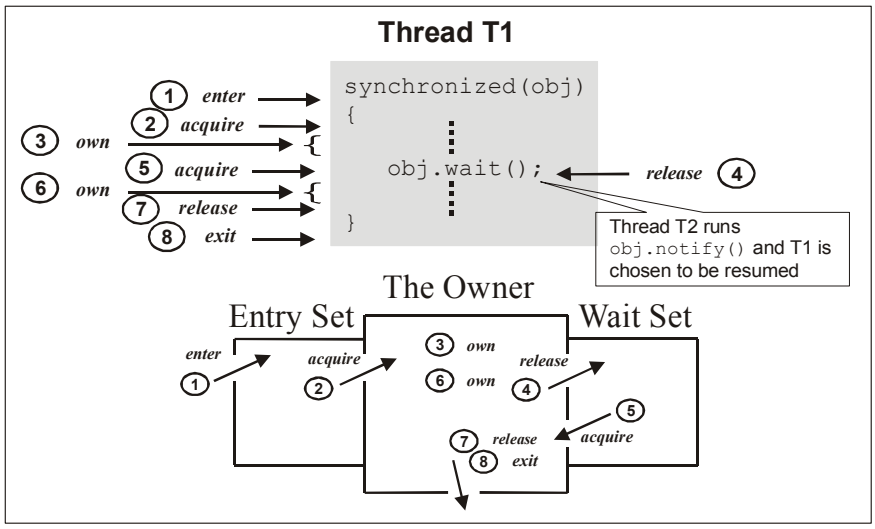

Figura 3-6. Secuencia de acciones sobre un monitor generadas por una llamada al método `wait`. (Basada en la ilustración de monitor Java, Bill Venders [2]).

Para el caso de la llamada a `notify` por parte de un thread (sea T2 dicho thread), se pasa por las siguientes acciones (ver **Figura 3-7**):

- Cuando T2 llega al inicio del syncrhonized block compite con los demás threads para adquirir el lock (**1. enter**).
- Una vez ha adquirido el lock (**2. acquire**), T2 pasa a poseer el monitor (**3. own**).
- T2 empieza a ejecutar la región del monitor y llega al punto de la llamada a `notify`.
- La ejecución de `notify` hace que T2 elija a un thread del wait set para que continúe su ejecución (sea T1 el thread elegido).
- T2 continúa la ejecución hasta que libera al lock (**4. release**) y sale del monitor (**5. exit**).

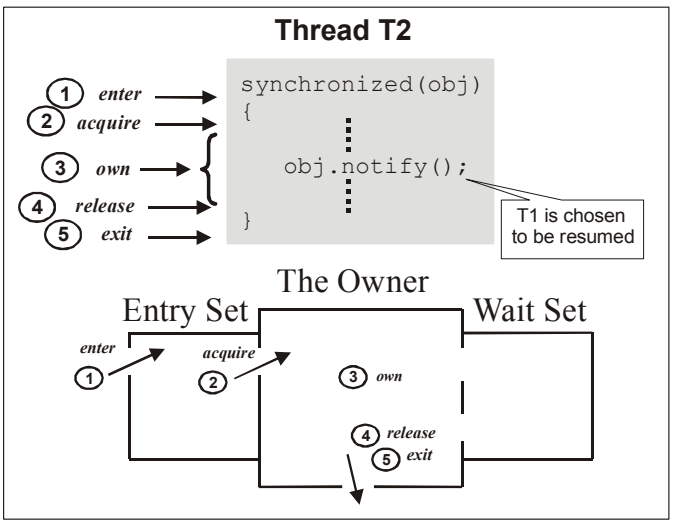

Figura 3-7. Secuencia de acciones sobre un monitor generadas por una llamada al método `notify`. (Basada en la ilustración de monitor Java, Bill Venders [2]).

Al igual que en otros lenguajes de programación que habilitan el uso de concurrencia, Java no queda libre de los problemas achacables a la programación concurrente[5]. Básicamente estos problemas se pueden resumir en los siguientes:

- **Race conditions** (o condiciones de carrera): Varios threads intentan actualizar al mismo tiempo la misma estructura de datos pudiendo ocasionar una inconsistencia en los datos contenidos en la estructura de datos.

- **Deadlock** (o abazo mortal): Dos o más threads entran en una situación de abrazo mortal cuando un thread A tiene una reserva sobre el recurso X y quiere acceder al recurso Y. A su vez, otro thread B tiene una reserva sobre el recurso Y y quiere acceder al recurso X.

- **Starvation** (o inanición): Se dice que un thread está en estado de inanición cuando es un thread ejecutable (en estado RUNNABLE) pero nunca es elegido por el scheduler, debido a que existen threads con más prioridad de forma que el thread no hace progreso en su ejecución.

4. Implementación de la JVM: Análisis del código fuente de Java 2 SDK

Vista la estructura interna de la JVM, seguiremos viendo la implementación que realiza Sun Microsystems de la especificación de la JVM. Partiendo de la distribución de código fuente del Java 2 SDK[6], vamos a analizar la implementación describiendo los módulos y abstracciones que se realicen en ella (centrándonos en lo referente a la implementación de elementos de sincronización y threads).

Se distinguen los siguientes niveles de implementación en el código fuente:

- **Java classes**: En este primer nivel se encuentra la primera parte de la implementación del API de Java. La implementación está escrita en Java, común para todas las plataformas y, por tanto, independiente de la plataforma de ejecución.

- **Java native**: En este segundo nivel se haya la segunda parte (y última) de la implementación del API de Java. Esta capa está implementada en C y C++, parte de la cual se genera en tiempo de compilación del código fuente del Java 2 SDK. Es en este nivel donde se implementan los métodos nativos de cada clase del API de Java, vinculando una tabla de traducción entre los nombres de los métodos nativos y nombres de funciones que pertenecen al nivel inferior (JNI[7]-JVM).

- **JNI – JVM**: En este tercer nivel se da acceso al interfaz nativo de Java (JNI), formado por un conjunto de funciones que ofrecen un API en tiempo de ejecución para interactuar con la JVM. Este interfaz viene definido a través de los ficheros de cabecera ubicados en el siguiente path,

[5] En el artículo de Allen Holub [10] se analizan estas situaciones en Java, documentándolas con ejemplos de código.
[6] Versión 1.2.2_006, para Linux, Solaris y Win32.
[7] Del inglés Java Native Interface. Es la implementación del intefaz definido en la especificación como Native Method Interface.

relativo a la localización de los fuentes de la implementación del Java 2 SDK: `src/share/javavm/export/` En este directorio se encuentran los siguientes ficheros:

Fichero	Descripción
jni.h	Java Native Interface: Donde se define el conjunto de funciones y estructuras para acceder a las operaciones internas de la JVM.
jvm.h	Java Virtual Machine: Ofrece funciones complementarias al interfaz ofrecido por `jni.h` Ofrece funciones para librerías nativas (por ejemplo, para implementar clases del API de Java tal como `java.lang.Object`), para dar soporte al Verificador y al Comprobador de los Ficheros de Clases y funciones de Entrada/Salida y de red.
jvmdi.h	Java Virtual Machine Debug Interface: Define un conjunto de funciones para que la JVM pueda ejecutarse en modo debug.
jvmpi.h	Java Virtual Machine Profiler Interface: Interfaz para poder obterner medidas de rendimiento de la JVM

Tabla 4-1. Interfaces ofrecidos por el nivel JNI-JVM para dar acceso a sus funcionalidades.

- **HPI**[8]: En este cuarto nivel se define un interfaz para realizar una abstracción de la plataforma de ejecución. Es en esta capa donde se realizan las llamadas al sistema operativo y donde se determina el tipo de threads a los que va a dar soporte la JVM (green threads o native threads).

La **Figura 4-1** muestra la disposición de los niveles de implementación de Java 2 SDK y la ubicación del código fuente de cada nivel. Es en el nivel HPI donde se encuentra el mayor grado de dependencia con la plataforma. Los threads de Java se implementan mediante funciones de librerías de threads (Pthreads para Linux y Pthreads o Solaris Threads para Solaris) o, directamente, por llamadas al sistema operativo que dan soporte a gestión de threads (para el caso de Win32 o el modelo de green threads, que se apoya en `fork` para crear al proceso que alberga a los threads de usuario).

[8] Del inglés Host Porting Interface.

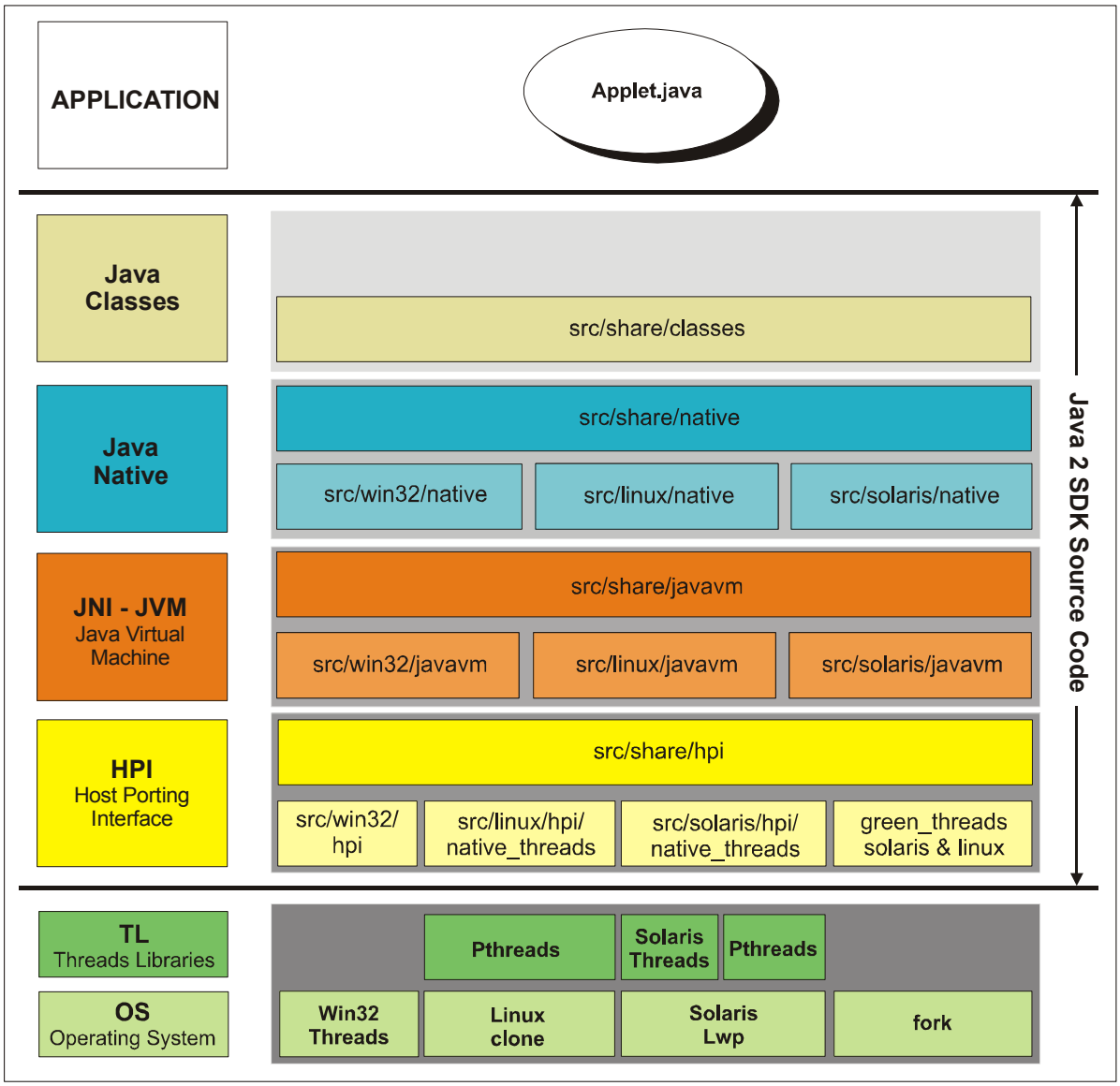

Figura 4-1. Disposición de los niveles de implementación de Java 2 SDK (Java Classes – JC, Java Native – JN, JNI-JVM y HPI) junto a la ubicación del código fuente de cada nivel.

Las dos figuras que vienen a continuación ilustran el esquema de implementación de Java 2 SDK, viendo las diferentes capas de implementación e interfaz de llamadas que ofrece cada una:

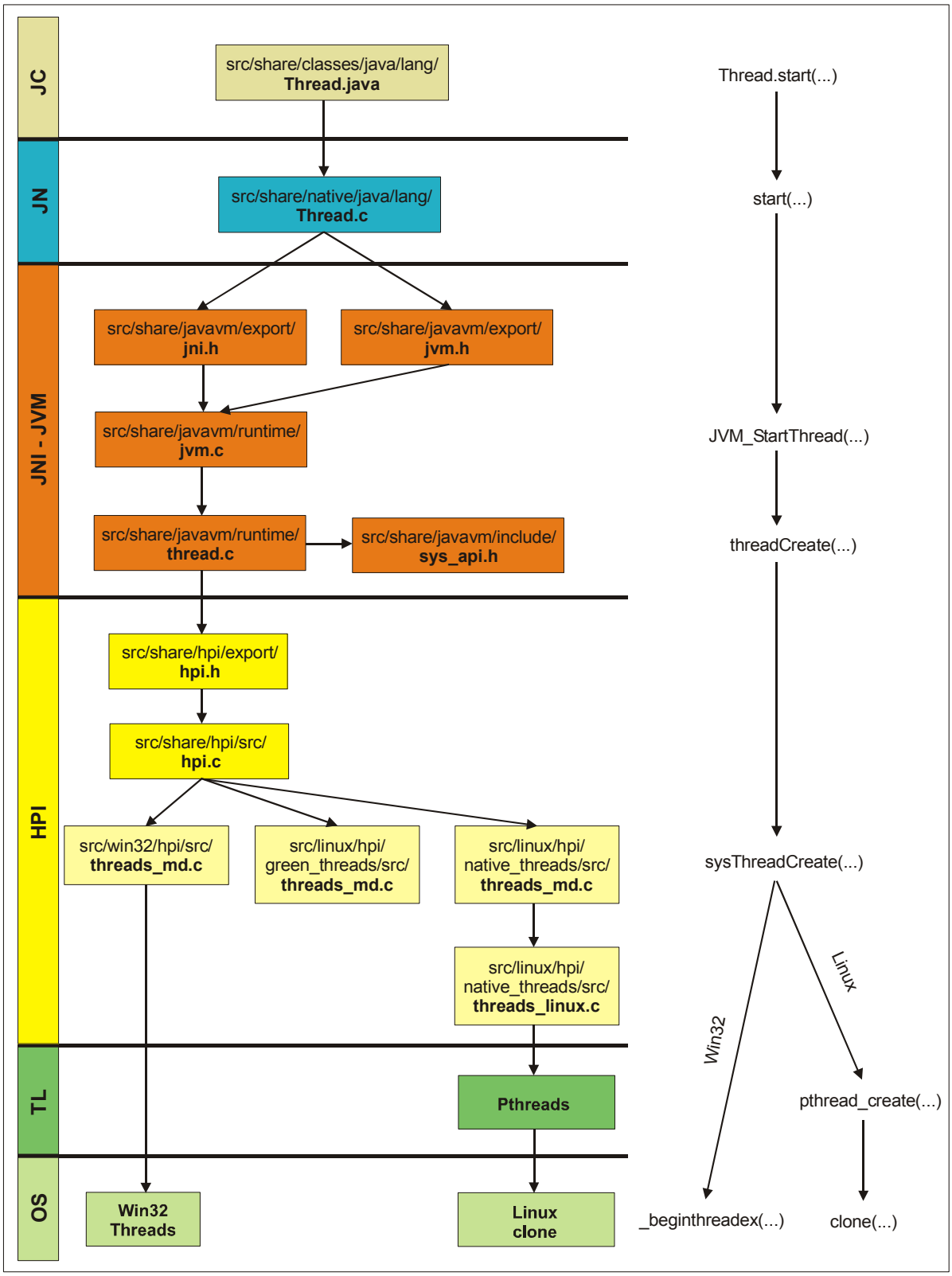

Figura 4-2. Secuencia de llamadas que se generan al realizar una llamada al método `start()` de un objeto de la clase `Thread`, para los modelos de Green Threads (Linux) y Native Threads (Linux y Win32).

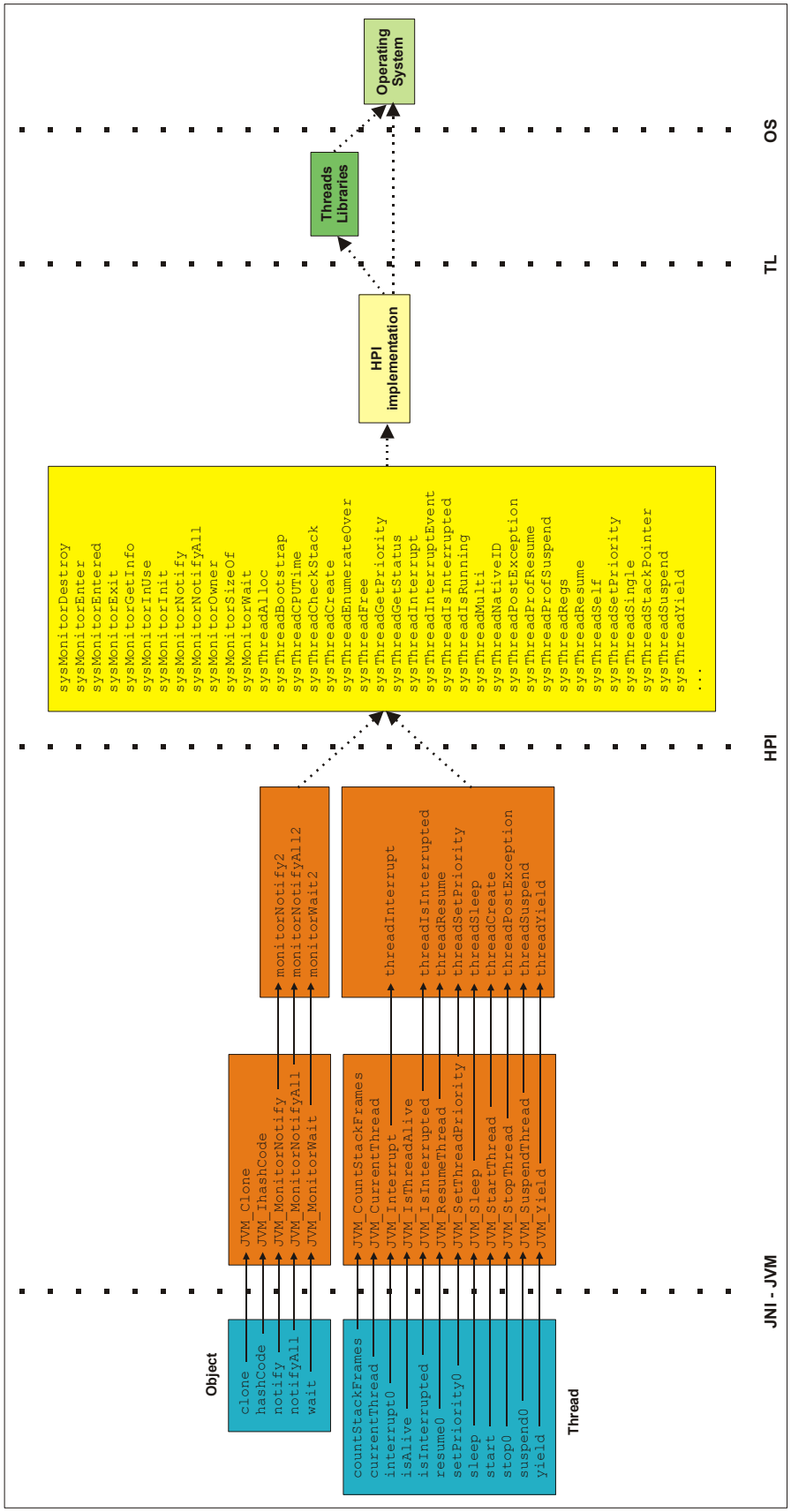

Figura 4-3. Correspondencia de llamadas entre los niveles de implementación.

La **Figura 4-2** ilustra la dependencia de llamadas por capas que tiene una llamada a un método de un objeto de la clase Thread. En este caso, se trata del método start, que es un método nativo y su implementación (en el nivel HPI) variará en función del modelo de threads que vaya a dar soporte la JVM. Se puede observar que en el caso del modelo de green threads, la implementación finaliza en el nivel HPI sin hacer ninguna llamada a librería (o directamente a sistema operativo) con relación a gestión de threads. En cambio, en el modelo de native threads, se ve como la implementación del nivel HPI realiza llamadas a librería (en caso de Linux) o a sistema operativo (caso Win32) para usar las funcionalidades de threads.

La segunda (**Figura 4-3**) es un esquema de llamadas común para todas las implementaciones de la Máquina Virtual de Java, es decir, el interfaz de llamada hasta la capa HPI es común, independientemente del tipo de threads que vaya a dar soporte la JVM e independiente del sistema operativo subyacente. Las llamadas correspondientes a los métodos nativos, dentro del nivel Java Native (JN), se corresponden una a una con una función del nivel JNI-JVM. Dicha relación se implementa a través de una tabla de traducción entre nombre de función nativa (por ejemplo, `resume0`) y nombre de función del nivel JNI-JVM (`JVM_ResumeThread`). En el nivel JNI-JVM existen funciones cuya implementación no requiere realizar llamadas al nivel HPI y otras sí. No es hasta el nivel HPI donde la implementación se hace dependiente de la plataforma de ejecución.

Llegados a este punto, vamos a analizar cada nivel de implementación de Java 2 SDK viendo los diferentes elementos que ofrece y cómo los implementa.

4.1. JC (Java Classes)

En este nivel se encuentra la implementación en lenguaje Java del API de Java. Más concretamente, es el primer nivel de implementación del API de Java realizado en su totalidad en lenguaje Java. A partir del directorio, del código fuente de Java 2 SDK, `src/share/classes` se encuentran organizadas por paquetes todas las clases que forman el API de Java.

Centrando la atención en el código de las clases `java.lang.Object` y `java.lang.Thread`, obtenemos la siguiente información:

- Toda clase dispone de un método nativo llamado registerNatives que es llamado automáticamente cuando el cargador de clases carga la clase.
- Aparece la declaración de métodos nativos sin código Java asociado.
- En tiempo de compilación, por cada clase se genera un fichero de cabecera (por ejemplo, de `Thread.java` se genera `java_lang_Thread.h`) que contendrá la traducción de Java a C de dicha clase. Este fichero de cabecera contiene la declaración de tipos y de funciones correspondientes a la implementación de los métodos nativos.

Entre otros tipos, en este nivel se declaran los siguientes tipos:

Tipo	Declarado en
Classjava_lang_Thread	build/win32/java/jvm/CClassHeaders/java_lang_Thread.h
Classjava_lang_ThreadGroup	build/win32/java/jvm/CClassHeaders/java_lang_ThreadGroup.h
Hjava_lang_Thread	build/win32/java/jvm/CClassHeaders/java_lang_Thread.h
Hjava_lang_ThreadGroup	build/win32/java/jvm/CClassHeaders/java_lang_ThreadGroup.h

Tabla 4-2. Tipos declarados en el nivel Java Classes (JC), tomando como plataforma de referencia a Win32.

4.2. JN (Java Native)

En este nivel se encuentra la segunda parte de la implementación del API de Java realizado en lenguaje C y C++. A partir del directorio `src/share/native` se encuentran organizadas por paquetes todas las clases que forman el API de Java.

Principalmente, este nivel realiza la función de traducción entre nombres de métodos nativos de Java a nombres de funciones del nivel JNI-JVM. Así pues, para las clases tenemos las siguientes declaraciones de tablas de traducción de métodos nativos (cabe decir que la clase `java.lang.ThreadGroup` no tiene métodos nativos):

Clase	Tabla de traducción

Object	`static JNINativeMethod methods[] = {`
	`  {"hashCode",  "()I",  (void *)&JVM_IHashCode},`
	`  {"wait",  "(J)V",  (void *)&JVM_MonitorWait},`
	`  {"notify",  "()V",  (void *)&JVM_MonitorNotify},`
	`  {"notifyAll",  "()V",  (void *)&JVM_MonitorNotifyAll},`
	`  {"clone",  "()Ljava/lang/Object;",  (void *)&JVM_Clone},`
	`};`

Thread	`static JNINativeMethod methods[] = {`
	`  {"start",  "()V",  (void *)&JVM_StartThread},`
	`  {"stop0",  "(" OBJ ")V",  (void *)&JVM_StopThread},`
	`  {"isAlive",  "()Z",  (void *)&JVM_IsThreadAlive},`
	`  {"suspend0",  "()V",  (void *)&JVM_SuspendThread},`
	`  {"resume0",  "()V",  (void *)&JVM_ResumeThread},`
	`  {"setPriority0",  "(I)V",  (void *)&JVM_SetThreadPriority},`
	`  {"yield",  "()V",  (void *)&JVM_Yield},`
	`  {"sleep",  "(J)V",  (void *)&JVM_Sleep},`
	`  {"currentThread",  "()" THD,  (void *)&JVM_CurrentThread},`
	`  {"countStackFrames",  "()I",  (void *)&JVM_CountStackFrames},`
	`  {"interrupt0",  "()V",  (void *)&JVM_Interrupt},`
	`  {"isInterrupted",  "(Z)Z",  (void *)&JVM_IsInterrupted},`
	`};`

Tabla 4-3. Tablas de traducción, para las clases `java.lang.Object` y `java.lang.Thread`, de métodos nativos a funciones del nivel JNI-JVM.

4.3. JNI – JVM (Java Native Interface – Java Virtual Machine)

JNI [1] es un interfaz, definido en `src/share/javavm/export/`**`jni.h`**, que ofrece la posibilidad de poder desarrollar aplicaciones que combinen código escrito en Java, que se ejecuta dentro de una JVM, con otro código escrito en otros lenguajes (como por ejemplo C++).

Principalmente, el API JNI es usado para implementar métodos nativos de Java para gestionar situaciones en las cuales el lenguaje Java no puede. Por ejemplo: disponemos de una librería implementada con otro lenguaje y queremos usar sus funciones en programas escritos en

Java. Otro ejemplo sería la necesidad de implementar una parte de código de una aplicación lo más optimizado posible en coste de ejecución (temporal), con lo cual sería más idóneo desarrollarla con un lenguaje de más bajo nivel como el lenguaje ensamblador.

Una característica importante de este interfaz, usado en conjunción con el **Invocation API** (API de Invocación), es permitir incorporar la Máquina Virtual de Java dentro de una aplicación.

También, programando usando el JNI, es posible escribir métodos nativos para:

- Crear, inspeccionar y actualizar objetos Java.
- Llamar a métodos Java.
- Tratar y lanzar excepciones.
- Cargar y obtener información de clases.
- Realizar comprobación de tipos en tiempo de ejecución.

Por otro lado, mediante el API JVM (definido en el archivo `src/share/javavm/export/`**`jvm.h`**) se definen un conjunto de funciones que son complementarias a las ofrecidas por el JNI. Como se indica en este archivo, estas funciones se pueden agrupar en 3 grupos:

- Funciones relacionadas con la Máquina Virtual necesitadas por las librerías nativas en el API estándar de Java (la clase `java.lang.Object` necesita funciones a nivel de Máquina Virtual que esperen (`wait`) y notifiquen (`notify`) a threads en monitores).
- Funciones y constantes usadas por el verificador de bytecode (bytecode verifier) y por el comprobador de formato de fichero de clase (class file format checker).
- Y por último, funciones que implementan operaciones de entrada/salida y red usadas por los APIs estándar de red y de entrada/salida.

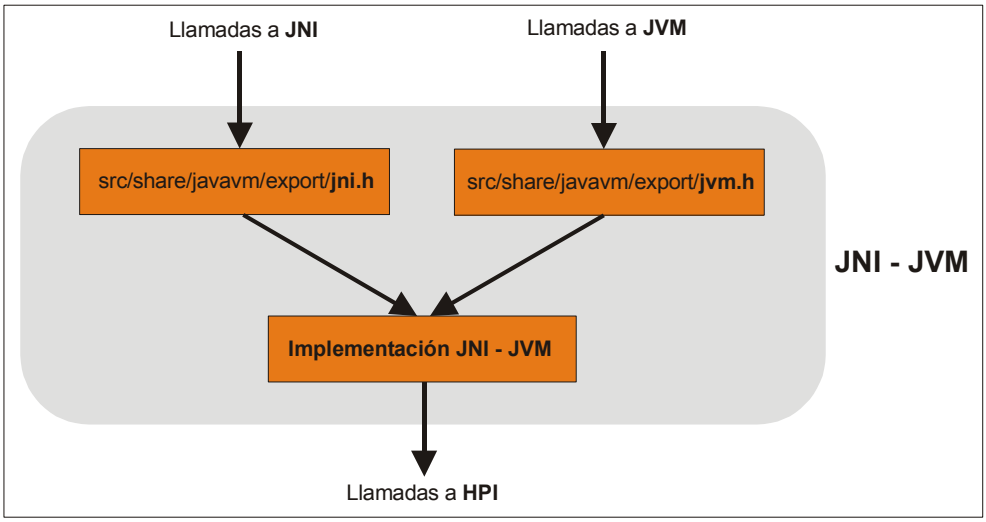

Figura 4-4. Estructura interna del nivel JNI – JVM.

En este nivel, los tipos declarados más relevantes son:

Tipo	Declarado en
`Classjava_lang_Object`	`src/share/javavm/include/oobj.h`
`ExecEnv`	`src/share/javavm/include/interpreter.h`
`Hjava_lang_Object`	`src/share/javavm/include/oobj.h`
`JNIEnv`	`src/share/javavm/export/jni.h`
`monitor_t`	`src/share/javavm/include/monitor.h`
`reg_mon_t`	`src/share/javavm/include/monitor.h`
`struct methodtable`	`src/share/javavm/include/oobj.h`

Tabla 4-4. Tipos declarados en el nivel JNI-JVM.

4.4. HPI (Host Porting Interface)

El interfaz HPI es el encargado de conectar la Máquina Virtual de Java con el sistema operativo donde se va a ejecutar. Es decir, HPI es el encargado de implementar una abstracción de la plataforma de ejecución a través de un conjunto de subsistemas que serán los encargados de implementar el acceso a los recursos gestionados por el sistema operativo.

Así, HPI es el nivel de implementación de la JVM más dependiente de la plataforma de ejecución. Para hacer la implementación de HPI más independiente, HPI ofrece un conjunto de interfaces para dar un acceso más uniforme a la implementación de sus funcionalidades. En la siguiente tabla se recogen dichos interfaces, definidos en `src/share/hpi/export/`**`hpi.h`**:

Interfaz	Función
Memoria	Gestión de la memoria
Librería	Soporte de librerías de enlace dinámico (DLL)
Sistema	Información de la plataforma hardware de ejecución y gestión de signals
Threads	Gestión de threads
Ficheros	Gestión de entrada/salida
Sockets	Comunicación a través de red vía sockets

Tabla 4-5. Conjunto de interfaces ofrecidos por el nivel HPI que dan acceso a la implementación de la abstracción de la plataforma de ejecución.

En la implementación del interfaz HPI se realizan llamadas al sistema operativo, vinculando así la implementación a un sistema operativo en concreto (Linux, Windows, Solaris...). Como consecuencia, es en esta fase donde se hace elección del modelo de threads que dará soporte la JVM (**green threads** o **native threads**). En el caso de Green Threads, sólo está disponible para Linux y Solaris (la llamada a `fork` de la **Figura 4-5** representa la existencia de un único proceso que gestiona los green threads – threads de usuario). Y el modelo de Native Threads está disponible para Win32 (se implementa haciendo llamadas directamente al API Win32), Linux (a través de la librería Pthreads, que usa la llamada al sistema clone de Linux) y Solaris (a través de la librería Solaris Threads o Pthreads, ambas implementadas usando el API de llamadas al sistema llamado Solaris LWP).

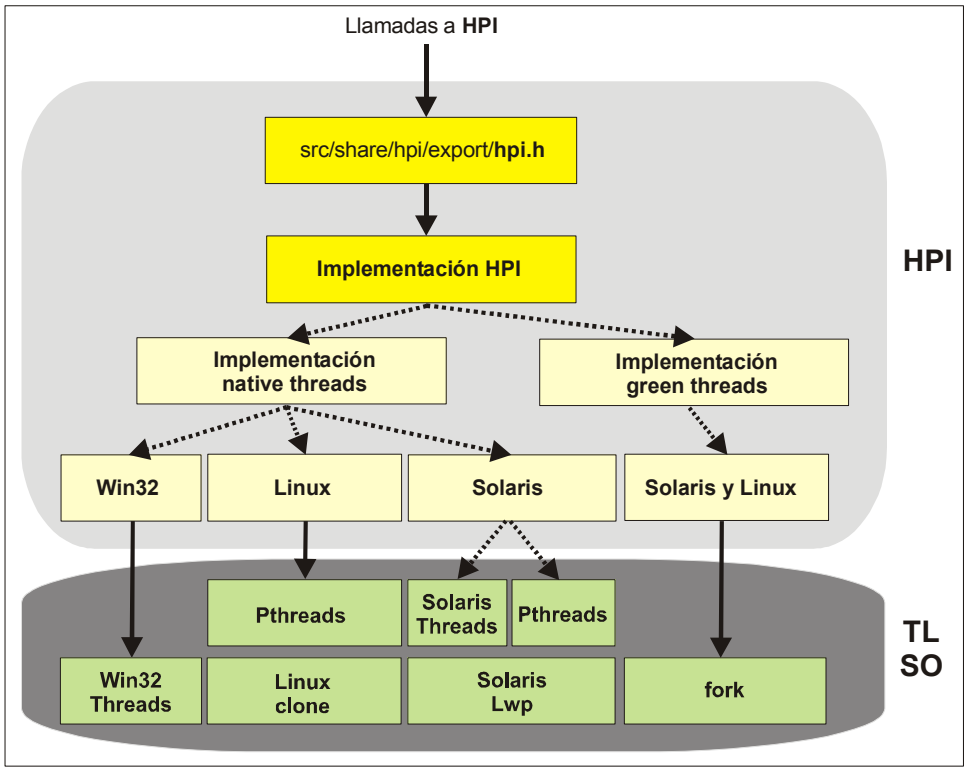

Figura 4-5. Implementación de HPI en función de los modelos de threads.

5. Dependencia entre HPI y la plataforma de ejecución

Veamos ahora una serie de ejemplos que nos permitirán comprender mejor el nivel de dependencia existente entre la implementación de HPI y la plataforma. Para ello, mostraremos la implementación de varias funciones del interfaz HPI comentando las peculiaridades del código para cada plataforma.

Se han elegido 2 funciones (`sysThreadCreate` y `sysThreadSetPriority`) para ilustrar las diferencias existentes entre las implementaciones de HPI en Linux, Solaris y Win32. Previo al análisis de dichas funciones, en la siguiente tabla definimos una serie de atributos que describen las características de un thread:

Atributo	Descripción
ContentionScope	Determina si un thread está asociado (bound) o no (unbound) a una entidad planificable por el sistema operativo.
DetachState	Indica si se guarda o no el estado del thread cuando éste finaliza su ejecución (si no se crea detached, se dice que es un thread joinable).
InheritSched	Especifica si el thread hereda o no la política de planificación del thread creador.
Priority	Prioridad con la que se crea el thread.
SchedPolicy	Expresa la política de planificación que se debe usar con dicho thread.
StackAddr	Dirección de la pila del thread.
StackSize	Tamaño de la pila del thread.

Tabla 5-1. Atributos de un thread.

El análisis de las llamadas es el siguiente:

sysThreadCreate

Prototipo	`int`
	`sysThreadCreate`
	`(sys_thread_t **tidP, long ss, void (*start)(void *), void *arg)`

Descripción	Crea un thread[9] en tidP, con una pila de ss bytes, que ejecutará el procedimiento referenciado por start (procedimiento que recibe un puntero como parámetro) pasándole arg como argumento. El thread se crea suspendido (en estado SUSPENDED). La llamada devuelve SYS_OK si se ha creado correctamente el thread, en caso contrario devuelve un entero indicando el error.

Pseudocódigo	`{`
	`  sys_thread_t * tid; /* Para el nuevo thread */`
	`  int err;  /* Gestión de errores */`
	
	`  /* Reserva memoria para el nuevo thread a crear */`
	
	`  tid = asigna_memoria(sys_thread_t);`
	
	`  gestion_error(tid);`
	
	`  /* Inicializa los campos de tid */`
	
	`  tid->estado = SUSPENDED; /* Estado inicial del thread */`
	`  tid->codigo = start;  /* Función que ejecutará el thread */`
	`  tid->parametro = arg;  /* Parámetro que se le pasará a la`
	`  función del thread */`
	
	`  /* Realiza la creación del thread vía la función crea_thread`
	`  que representa la llamada a una función de threads de`
	`  librería o a una llamada al sistema. */`
	
	`  err = crea_thread(tid->tid_del_sistema, ...);`
	
	`  gestion_error(err);`
	
	`  /* Realiza las acciones necesarias para que el nuevo thread`
	`  se cree en estado SUSPENDED */`
	
	`  err = suspende_thread(tid->tid_del_sistema);`
	
	`  gestion_error(err);`
	
	`  /* Agrega el nuevo thread a la cola global de threads */`
	
	`  encolar(ColaGlobalThreads, tid);`
	
	`  return err;`
	`}`

Tabla 5-2. Descripción de `sysThreadCreate`.

Linux:

La creación de un nuevo thread se realiza mediante la llamada `pthread_create` del API LinuxThreads (POSIX[10] threads o Pthreads), que crea un Pthread. El Pthread se crea con los siguientes atributos:

[9] Thread ofrecido por la plataforma en concreto (Pthreads, Win32 threads, Solaris threads, etc).
[10] Estándar IEEE POSIX (Portable Operating System Interface) 1003.1c thread API.

Atributo	Descripción
ContentionScope	Por defecto, `PTHREAD_SCOPE_SYSTEM`. (`PTHREAD_SCOPE_PROCESS` no es soportado por LinuxThreads). Siempre un Pthread está asociado a una entidad planificable por el sistema (thread bound).
DetachState	`PTHREAD_CREATE_DETACHED` (no se crea joinable).
InheritSched	Por defecto, `PTHREAD_EXPLICIT_SCHED`.
Priority	Prioridad por defecto, 0.
SchedPolicy	Por defecto, `SCHED_OTHER` (regular, non-realtime scheduling o timesharing).
StackAddr	Por defecto, la asignada por el sistema.
StackSize	No modificable en LinuxThreads.

Tabla 5-3. Atributos de creación de un POSIX thread en Linux.

Como el nuevo thread debe crearse en estado `SUSPENDED` y los Pthreads creados por `pthread_create` en Linux se crean en estado `RUNNABLE`, se recurre al uso de semáforos POSIX[11] para poder suspender al Pthread.

Solaris:

En Solaris existen 2 posibles implementaciones según la librería de threads elegida (POSIX threads o Solaris threads):

- POSIX threads:

Para crear el nuevo thread se realiza una llamada a `pthread_create`, creando un thread con los siguientes atributos:

Atributo	Descripción
ContentionScope	Por defecto, `PTHREAD_SCOPE_PROCESS`. El nuevo Pthread no está asociado a una entidad planificable del sistema (en Solaris, LWP).
DetachState	`PTHREAD_CREATE_DETACHED` (no se crea joinable).
InheritSched	Por defecto, `PTHREAD_EXPLICIT_SCHED`.
Priority	Prioridad por defecto, 0.
SchedPolicy	Por defecto, `SCHED_OTHER` (regular, non-realtime scheduling o timesharing).
StackAddr	Por defecto, la asignada por el sistema.
StackSize	El tamaño pasado por parámetro. El tamaño por defecto es de 1 Mbyte para procesos de 32 bits y 2 Mbytes para de 64 bits[12].

Tabla 5-4. Atributos de creación de un POSIX thread en Solaris.

Los POSIX threads se crean en estado `RUNNABLE`, con lo que se debe cambiar su estado a `SUSPENDED`. Este cambio se realiza mediante el uso de variables de condición y mutexes.

- Solaris threads:

La creación de un nuevo thread se realiza de una forma más sencilla, realizando una única llamada a `thr_create`. El thread se crea con los siguientes atributos:

[11] Estándar IEEE POSIX 1003.1b semaphores API.
[12] En Solaris 8.

Atributo	Descripción
ContentionScope	Por defecto, sin el flag `THR_BOUND`. El nuevo thread no está asociado a una entidad planificable del sistema.
DetachState	`THR_DETACHED` (no se crea joinable).
Priority	Prioridad por defecto, 0.
SchedPolicy	Sólo se permite `SCHED_OTHER` (timesharing).
StackAddr	Por defecto, la asignada por el sistema.
StackSize	El tamaño pasado por parámetro. El tamaño por defecto es de 1 Mbyte para procesos de 32 bits y 2 Mbytes para de 64 bits.
Suspended flag	Flag activado, `THR_SUSPENDED` (por defecto, se crea `RUNNABLE`).

Tabla 5-5. Atributos de creación de un Solaris thread.

El thread se crea en estado `SUSPENDED` (se pasa el flag `THR_SUSPENDED`), con lo que no es necesario ningún código extra como en la versión de POSIX threads.

Win32:

En la implemantación Win32, el nuevo thread se crea efectuando una llamada a `_beginthreadex` con los siguientes atributos:

Atributo	Descripción
ContentionScope	El thread es una entidad planificable por el sistema (thread bound).
Priority	Por defecto, `THREAD_PRIORITY_NORMAL`.
StackSize	El proporcionado por el parámetro (por defecto, el tamaño de la pila del thread principal – el primer thread del proceso).
Suspended flag	Flag activado, `CREATE_SUSPENDED` (por defecto el thread se crea `RUNNABLE`). El thread se crea en estado `SUSPENDED`.

Tabla 5-6. Atributos de creación de un Win32 thread.

`sysThreadSetPriority`

Prototipo	`int`
	`sysThreadSetPriority`
	`(sys_thread_t * tid, int pri)`

Descripción	Establece la prioridad del thread tid a pri. La llamada devuelve `SYS_OK` si se ha asignado correctamente la nueva prioridad, en caso contrario devuelve un entero indicando el error.

Pseudocódigo	`{`
	`  int err;  /* Gestión de errores */`
	
	`  /* Asigna la nueva prioridad del thread vía la función`
	`  asigna_prioridad que representa la llamada a una función`
	`  de threads de librería o a una llamada al sistema. */`
	
	`  err = asigna_prioridad(tid->tid_del_sistema, pri, ...);`
	
	`  gestion_error(err);`
	
	`  return err;`
	`}`

Tabla 5-7. Descripción de `sysThreadSetPriority`.

Linux:

La implementación se realiza usando una llamada a `pthread_setschedparam`. Dado que la política de planificación es `SCHED_OTHER`, una prioridad diferente a 0 no será tenida en cuenta[13]. De tal forma, la prioridad establecida en un thread de Java no tendrá efecto.

Solaris:

Al igual que en Linux, la implementación se realiza llamando a `pthread_setschedparam`. La implementación de Pthreads en Solaris se limita a una política de planificación (`SCHED_OTHER`) y la prioridad de un thread puede tomar un valor de 0 a 127. Como consecuencia, en Solaris sí se tiene en cuenta la prioridad establecida en un thread de Java. Para el caso de Solaris threads es análogo, variando solamente el interfaz.

Win32:

En Win32 se llama a `SetThreadPriority` para cambiar la prioridad de un thread de Win32. La prioridad asociada a un Win32 thread puede tomar 7 valores diferentes, así que la implementación que se realiza de la gestión de prioridades requiere hacer una correspondencia entre los 10 posibles valores de prioridad que puede tomar un thread de Java con los 7 de los threads de Win32. Dicha correspondencia ocasiona que 2 threads de Java con diferente prioridad, en Win32 tengan la misma prioridad (ver código de la **Tabla 5-13**).

En la **Figura 5-1** se muestra la relación existente entre las prioridades definidas por Java (nivel JVM) y las ofrecidas por el nivel HPI para las plataformas Green Threads (Linux y Solaris), Solaris Native Threads, Linux Native Threads y Win32 Native Threads, respectivamente.

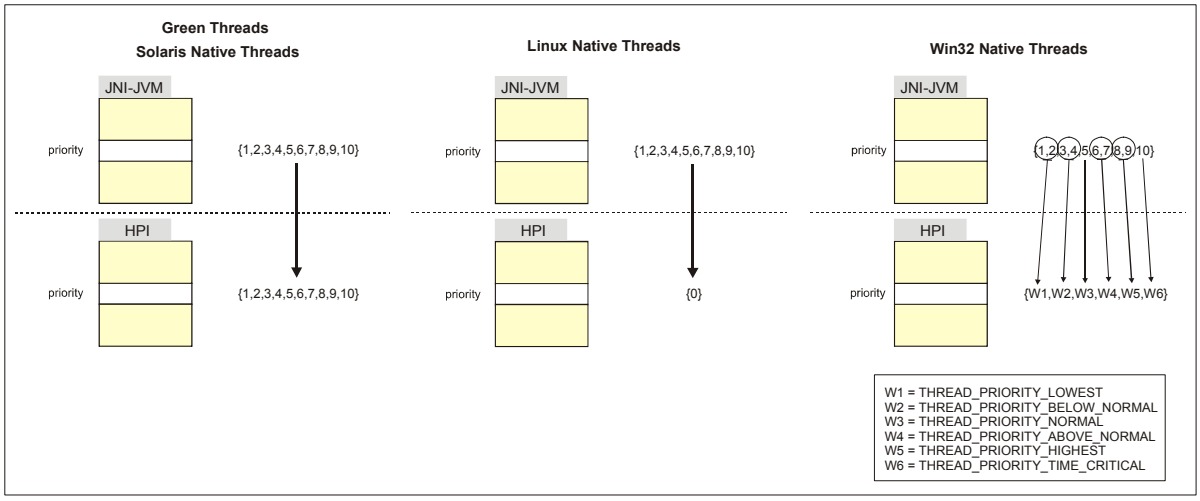

Figura 5-1. Mapeo de prioridades de JVM a HPI.

[13] En el código fuente que implementa `pthread_setschedparam`, se ve claramente en la sentencia: `th->p_priority = policy == SCHED_OTHER ? 0 : param->sched_priority;`

Implementación de `sysThreadCreate`

Plataforma: Linux Native Threads

```
Int
sysThreadCreate(sys_thread_t **tidP, long ss, void (*start)(void *), void
*arg)
{
    size t stack size = ss;
    int err;
    sys_thread_t *tid = allocThreadBlock();
    pthread_attr_t attr;

    if (tid == NULL) {
        return SYS_NOMEM;
    }
    *tidP = tid;

#if 1
    memset((char *)tid, 0, sizeof(sys_thread_t));
#endif

    /* Install the backpointer to the Thread object */

    tid->interrupted = tid->pending interrupt = FALSE;
    tid->onproc = FALSE;
    tid->start proc = start;
    tid->start_parm = arg;
    tid->state = SUSPENDED;

#ifndef HAVE GETHRVTIME
    tid->last tick = (clock t) 0;
    tid->ticks = (jlong) 0;
#endif

    tid->primordial thread = 0;

    /* Semaphore used to block thread until np suspend() is called */
    err = sem init(&tid->sem ready to suspend, 0, 0);
    sysAssert(err == 0);
    err = sem_init(&tid->sem_suspended, 0, 0);
    sysAssert(err == 0);
    /* Thread attributes */
    pthread attr init(&attr);
#ifdef  POSIX THREAD ATTR STACKSIZE
    pthread_attr_setstacksize(&attr, stack_size);
#endif
    pthread attr setdetachstate(&attr, PTHREAD CREATE DETACHED);
    if (profiler_on) {
        pthread attr setscope(&attr, PTHREAD SCOPE SYSTEM);
    }
    /* Create the thread. The thread will block waiting to be suspended */
    err = pthread_create(&tid->sys_thread, &attr, _start, (void *)tid);
    sysAssert(err == 0);
    if (err == 0) {
        err = sem wait(&tid->sem ready to suspend);
        if (err == 0) {
            sem_destroy(&tid->sem_ready_to_suspend);
            /* Suspend the thread */
            err = np suspend(tid);
            if (err == SYS_OK) {
                /* Unblock the thread now that it ha
                err = sem_post(&tid->sem_suspended);
                sysAssert(err == 0);
            }
        }
    }
    sysAssert(err == 0);
```

En Linux no está implementada. No podemos especificar el tamaño de la pila de un Pthread

Creación del Pthread.

Suspende el thread vía semáforos POSIX (selfsuspend) o `pthread_kill`.

```
    SYS_QUEUE_LOCK(sysThreadSelf());
    ActiveThreadCount++;         /* Global thread count */
    tid->next = ThreadQueue;     /* Chain all threads */
    ThreadQueue = tid;
    SYS_QUEUE_UNLOCK(sysThreadSelf());

    sysAssert(err != EINVAL);    /* Invalid argument: shouldn't happen */
    if (err == EAGAIN) {
        err = SYS_NORESOURCE;    /* Will be treated as though SYS_NOMEM */
    } else if (err == ENOMEM) {
        err = SYS_NOMEM;
    } else {
        err = SYS_OK;
    }

    return err;
}
```

Tabla 5-8. Detalle de la función `sysThreadCreate` en Linux.

Plataforma: Solaris Native Threads

```
int
sysThreadCreate(sys_thread_t **tidP, long ss, void (*start)(void *), void *arg)
{
    size_t stack_size = ss;
    int err;
    sys_thread_t *tid = allocThreadBlock();
#ifdef USE_PTHREADS
    pthread_attr_t attr;
#endif

    if (tid == NULL) {
        return SYS_NOMEM;
    }
    *tidP = tid;

    /* Install the backpointer to the Thread object */

    tid->interrupted = FALSE;
    tid->onproc = FALSE;

    SYS_QUEUE_LOCK(sysThreadSelf());
    ActiveThreadCount++;         /* Global thread count */
    tid->next = ThreadQueue;     /* Chain all threads */
    ThreadQueue = tid;
    SYS_QUEUE_UNLOCK(sysThreadSelf());

    tid->start_proc = start;
    tid->start_parm = arg;

#ifdef USE_PTHREADS
    /* Setup condition required to suspend the newly created thread. */
    pthread_mutex_init(&tid->ntcond.m, NULL);
    pthread_cond_init(&tid->ntcond.c, NULL);
    tid->ntcond.state = NEW_THREAD_MUST_REQUEST_SUSPEND;
    pthread_mutex_lock(&tid->ntcond.m);

    /* Create the new thread. */
    pthread_attr_init(&attr);
#ifdef _POSIX_THREAD_ATTR_STACKSIZE
    pthread_attr_setstacksize(&attr, stack_size);
#endif
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    if (profiler_on)
        pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    err = pthread_create(&tid->sys_thread, &attr, _start, (void *)tid);

    /* Wait for the newly created thread to block. */
    while (tid->ntcond.state != NEW_THREAD_REQUESTED_SUSPEND)
```

En Solaris está implementada. Podemos especificar el tamaño de la pila de un Pthread.

Creación del Pthread.

```
        pthread_cond_wait(&tid->ntcond.c, &tid->ntcond.m);

    /* So it blocked.  Now suspend it and _then_ get it out of the block. */
    np_suspend(tid->sys_thread);
    tid->ntcond.state = NEW_THREAD_SUSPENDED;
    pthread_cond_signal(&tid->ntcond.c);
    pthread_mutex_unlock(&tid->ntcond.m);

#else
    /* Create the thread */
    err = thr_create(NULL, stack_size, _start, (void *)tid,
                     THR_SUSPENDED|THR_DETACHED|
                     (profiler_on ? THR_BOUND : 0),
                     &tid->sys_thread);
#endif /* USE_PTHREADS */

    tid->state = SUSPENDED;
    sysAssert(err != EINVAL);    /* Invalid argument: shouldn't happen */
    if (err == EAGAIN) {
        err = SYS_NORESOURCE;    /* Will be treated as though SYS_NOMEM */
    } else if (err == ENOMEM) {
        err = SYS_NOMEM;
    } else {
        err = SYS_OK;
    }

    return err;
}
```

Suspende el thread vía `thr_suspend`.

Creación del Solaris thread en estado `SUSPENDED`.

Tabla 5-9. Detalle de la función `sysThreadCreate` en Solaris.

Plataforma: Win32 Native Threads

```
/*
 * Create a new Java thread. The thread is initially suspended.
 */
int
sysThreadCreate(sys_thread_t **tidP, long stack_size,
                void (*proc)(void *), void *arg)
{
    sys_thread_t *tid = allocThreadBlock();
    if (tid == NULL) {
        return SYS_NOMEM;
    }
    tid->state = SUSPENDED;
    tid->start_proc = proc;
    tid->start_parm = arg;

    tid->interrupt_event = CreateEvent(NULL, TRUE, FALSE, NULL);

    /*
     * Start the new thread.
     */
    tid->handle = (HANDLE)_beginthreadex(NULL, stack_size, _start, tid,
                                         CREATE_SUSPENDED, &tid->id);
    if (tid->handle == 0) {
        return SYS_NORESOURCE;  /* Will be treated as though SYS_NOMEM */
    }

    queueInsert(tid);
    *tidP = tid;
    return SYS_OK;
}
```

Creación del Win32 thread en estado `SUSPENDED`.

Tabla 5-10. Detalle de la función `sysThreadCreate` en Win32.

Implementación de `sysThreadSetPriority`

Plataforma: Linux Native Threads

```
/*
 * Set the scheduling priority of a specified thread
 */
int
sysThreadSetPriority(sys_thread_t * tid, int pri)
{
    int err;
#ifdef USE SCHED OTHER
    struct sched_param param;
    param.sched_priority = pri;
    err = pthread setschedparam(tid->sys thread, SCHED OTHER, &param);
    sysAssert(err != ESRCH);      /* No such thread: shouldn't happen */
    sysAssert(err != EINVAL);     /* Invalid arguments: shouldn't happen */
#endif /* USE SCHED OTHER */
    return SYS_OK;
}
```

La asignación de una prioridad diferente de 0 en Linux, con `SCHED_OTHER`, no tiene efecto. La prioridad siempre es 0.

Tabla 5-11. Detalle de la función `sysThreadSetPriority` en Linux.

Plataforma: Solaris Native Threads

```
/*
 * Set the scheduling priority of a specified thread
 */
int
sysThreadSetPriority(sys_thread_t * tid, int pri)
{
    int err;
#ifdef USE PTHREADS
#ifdef USE_SCHED_OTHER
    struct sched_param param;
    param.sched priority = pri;
    err = pthread setschedparam(tid->sys thread, SCHED OTHER, &param);
#else
    err = 0;
#endif /* USE_SCHED_OTHER */
#else
    err = thr setprio(tid->sys thread, pri);
#endif /* USE PTHREADS */
    sysAssert(err != ESRCH);      /* No such thread: shouldn't happen */
    sysAssert(err != EINVAL);     /* Invalid arguments: shouldn't happen */
    return SYS_OK;
}
```

Asignación de la nueva prioridad para POSIX threads.

Asignación de la nueva prioridad para Solaris threads.

Tabla 5-12. Detalle de la función `sysThreadSetPriority` en Solaris.

Plataforma: Win32 Native Threads

```
/*
 * Set priority of specified thread.
 */
int
sysThreadSetPriority(sys_thread_t *tid, int p)
{
    int priority;

    switch (p) {
    case 0:
        priority = THREAD_PRIORITY_IDLE;
        break;
    case 1: case 2:
        priority = THREAD PRIORITY LOWEST;
        break;
```

Se da el caso de que para 2 prioridades diferentes de Java threads (casos: [1,2] , [3,4] , [6,7] y [8,9]) tienen asociada una única prioridad de Win32 threads.

```
    case 3: case 4:
        priority = THREAD_PRIORITY_BELOW_NORMAL;
        break;
    case 5:
        priority = THREAD PRIORITY NORMAL;
        break;
    case 6: case 7:
        priority = THREAD_PRIORITY_ABOVE_NORMAL;
        break;
    case 8: case 9:
        priority = THREAD PRIORITY HIGHEST;
        break;
    case 10:
        priority = THREAD_PRIORITY_TIME_CRITICAL;
        break;
    default:
        return SYS_ERR;
    }
    return SetThreadPriority(tid->handle, priority) ? SYS_OK : SYS_ERR;
}
```

Asignación de la nueva prioridad para Win32 threads.

Tabla 5-13. Detalle de la función `sysThreadSetPriority` en Win32.

Pensemos ahora en como se comportará un programa formado por varios threads con diferentes prioridades que sea ejecutado en diferentes plataformas Java. ¿Qué comportamiento ofrecerán las ejecuciones? Según las especificaciones de Java, un programa Java es independiente de la plataforma donde se ejecute, pero el análisis del código fuente de Java 2 SDK parece indicar lo contrario.

Veamos el grado de dependencia existente en la implementación del nivel HPI ante el comportamiento de la ejecución de un programa Java. Para ello, nos serviremos de una aplicación, representada en la **Figura 5-2**, cuyo thread principal crea 4 threads de diferentes prioridades que se ejecutarán concurrentemente y cuya finalización esperará el thread principal.

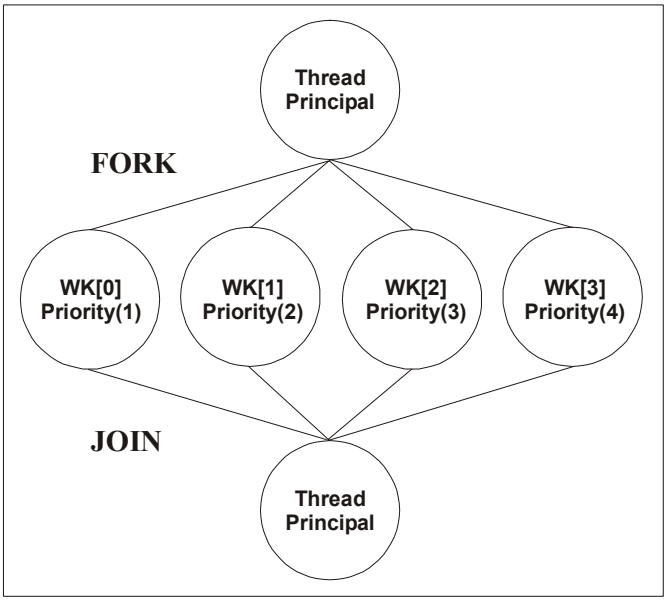

Figura 5-2. Modelo de ejecución fork-join a analizar.

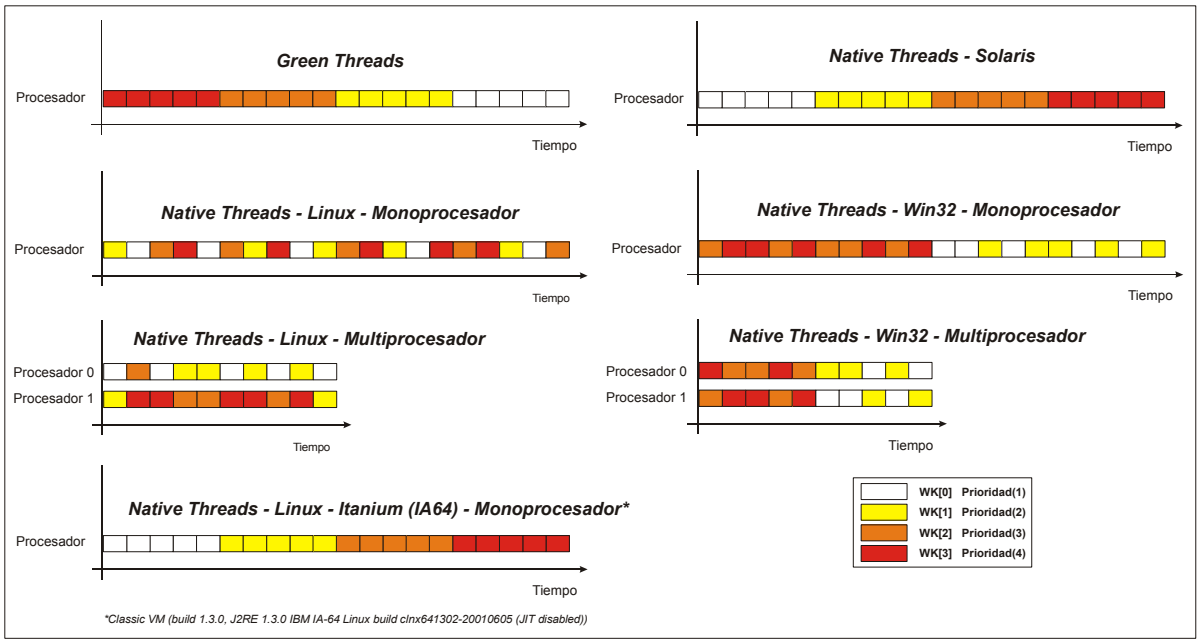

Figura 5-3. Resultados de la ejecución en diferentes plataformas.

En la **Figura 5-3** vemos el comportamiento de una ejecución de la aplicación en diferentes plataformas.

Green Threads

La ejecución de la aplicación se comporta como es de esperar, es decir, según la especificación de Java, los threads de Java se planifican mediante un algoritmo de scheduling por prioridades y preemptivo. Así pues, primero se ejecutará el thread más prioritario y por último el de menos prioridad. Tanto Green Threads para Linux o para Solaris se comportan idénticamente.

Linux Native Threads

El resultado nada tiene que ver con la ejecución teórica (según la especificación de Java). Es debido, principalmente a la política de planificación elegida (`SCHED_OTHER`) de los POSIX Threads en Linux. La prioridad de los Java threads no se tiene en cuenta siendo siempre 0, dejando al scheduler del kernel planificarlos en función de la prioridad dinámica (time-sharing). Por otro lado, la ejecución sobre una arquitectura Itanium (IA64, con una versión diferente de JVM) muestra un comportamiento totalmente diferente, siguiendo un scheduling FIFO.

Solaris Native Threads

En este caso, la ejecución se comporta idénticamente a la acaecida sobre la plataforma Linux Itanium (scheduling FIFO). Aparentemente la ejecución debería comportarse como la especificación de Java, pues la relación de prioridades y el algoritmo de scheduling es el correcto, pero en este caso los resultados no acompañan a la teoría.

Win32 Native Threads

La aplicación, al ejecutarse en esta plataforma se comporta como el código fuente nos ha mostrado: dos threads de Java con diferente prioridad se comportan, gracias al scheduler de Win32, como si tuvieran la misma prioridad.

6. Conclusiones y trabajo futuro

Con este estudio de la concurrencia de la JVM, hemos querido mostrar los mecanismos que ofrece Java como plataforma de desarrollo de aplicaciones concurrentes mediante los elementos disponibles a nivel de lenguaje de programación. De igual forma, hemos analizado cómo son implementados dichos mecanismos desde el punto de vista de la JVM y desde el punto de vista más dependiente de la plataforma (HPI), tomando como ejemplo de implementación de la JVM el ofrecido por Sun a través del Java 2 SDK versión 1.2.2-006.

Hemos comprobado los grados de dependencia de la ejecución de una misma aplicación Java en diferentes plataformas de ejecución, observando comportamientos desiguales ocasionados por el grado de dependencia existente entre el nivel HPI de la plataforma Java y el sistema operativo. Si bien el API de acceso que ofrece la JVM es independiente de la plataforma, la implementación del nivel HPI de la JVM hace que exista la posibilidad de que un mismo programa pueda comportarse de diferente forma dependiendo de la plataforma donde se ejecute.

Por tanto, la existencia de un HPI cuya semántica de sus operaciones varía en función de la plataforma para la que se implementa, a pesar de ofrecer un API común e independiente, ocasiona que la arquitectura de Java no sea tan independiente.

Como trabajo a realizar, proponemos la alternativa de implementar un nivel HPI que sea capaz de añadir un grado de abstracción con respecto a la plataforma de ejecución, de manera que usando un mismo HPI para diferentes plataformas consigamos minimizar los efectos de dependencia observados en este trabajo, consiguiendo así una política de gestión uniforme de las entidades de planificación ofrecidas por los distintos sistemas operativos.

Para lograr una semántica común de las operaciones para todas las plataformas Java, el HPI tendría que ofrecer unas entidades de planificación cuya gestión fuera independiente del sistema operativo. Una posible solución sería implementar una librería de threads de usuario que ofreciera un scheduling por prioridades y no limitada al modelo de threads M a 1 para poder aprovechar las arquitecturas multiprocesador.

Apéndice A: Clase java.lang.Thread

Una aplicación, escrita en Java, que deba usar los mecanismos de ejecución a través de threads se hará usando la clase Thread, que forma parte de la librería de clases estándar de Java, localizada en el paquete `java.lang`

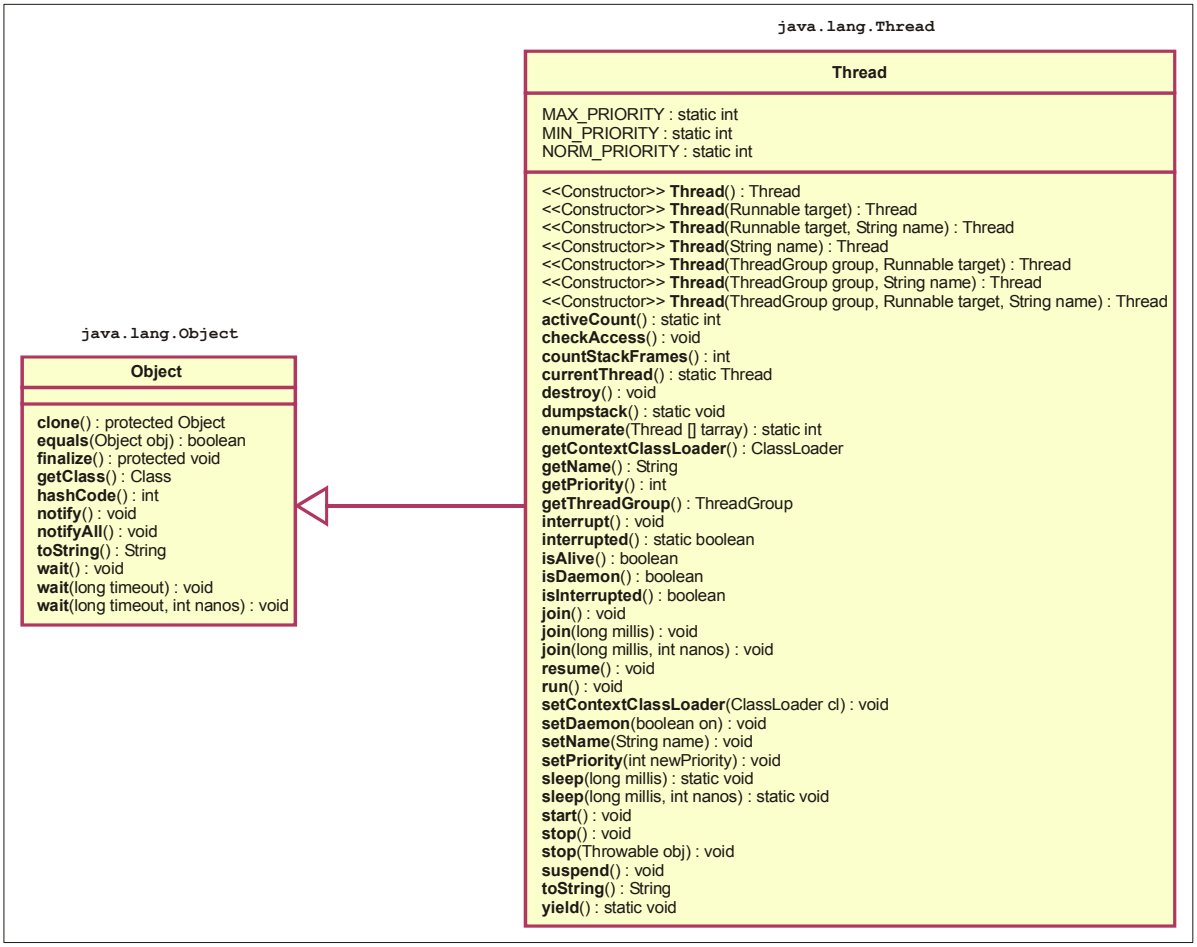

Figura Apéndice A-1. Diagrama de clases Object y Thread.

A continuación, describiremos el interfaz de programación proporcionado por la clase Thread, dando a conocer sus campos, constructores y métodos. Cabe decir que dentro de la jerarquía de clases estándar de Java, la clase `Thread` es descendiente de la clase `Object`, que es la clase a partir de la cual empieza la jerarquía.

Campos

MAX_PRIORITY

```
static int MAX_PRIORITY
```

La máxima prioridad que puede tener un thread.

MIN_PRIORITY

```
static int MIN_PRIORITY
```

La mínima prioridad que puede tener un thread.

NORM_PRIORITY

```
static int NORM_PRIORITY
```

La prioridad asignada por defecto a un thread.

Constructores

Thread

```
Thread()
Thread(Runnable target)
Thread(Runnable target, String name)
Thread(String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, String name)
Thread(ThreadGroup group, Runnable target, String name)
```

Crea un nuevo objeto Thread. El parámetro name indica el nombre del thread, target el objeto de ejecución y group al ThreadGroup al que ha de pertenecer.

Métodos

activeCount

```
static int activeCount()
```

Devuelve el número actual de threads activos en su grupo de threads.

checkAccess

```
void checkAccess()
```

Determina si el thread que se está ejecutando actualmente tiene permiso para modificar este thread.

countStackFrames

```
int countStackFrames()
```

DESCATALOGADA. La definición de esta llamada depende de `suspend()`, la cual está

descatalogada. Con lo cual, los resultados de esta llamada nunca estarán bien definidos.

currentThread

```
static Thread currentThread()
```

Devuelve una referencia al thread que se está ejecutando.

destroy

```
void destroy()
```

Destruye el thread sin ninguna limpieza.

dumpStack

```
static void dumpStack()
```

Imprime la traza de la pila del thread.

enumerate

```
static int enumerate(Thread[] tarray)
```

Copia en el vector de threads tarray cada thread activo del grupo y subgrupos de threads de este thread.

getContextClassLoader

```
ClassLoader getContextClassLoader()
```

Devuelve el contexto ClassLoader del thread.

getName

```
String getName()
```

Devuelve el nombre del thread.

getPriority

```
int getPriority()
```

Retorna la prioridad del thread.

getThreadGroup

```
ThreadGroup getThreadGroup()
```

Devuelve el grupo de threads al cual pertenece este thread.

interrupt

```
void interrupt()
```

Interrumpe la ejecución de este thread.

interrupted

```
static boolean interrupted()
```

Devuelve cierto o falso en función de si este thread ha sido interrumpido o no.

isAlive

```
boolean isAlive()
```

Devuelve cierto o falso en función de si el thread está vivo o no.

isDaemon

```
boolean isDaemon()
```

Indica si este thread es un demonio o no.

isInterrupted

```
boolean isInterrupted()
```

Indica si este thread ha sido o no interrumpido.

join

```
void join()
void join(long millis)
void join(long millis, int nanos)
```

Espera a que este thread muera.
Espera como mucho millis milisegundos a que este thread muera.
Espera como mucho millis milisegundos más nanos nanosegundos a que este thread muera.

resume

```
void resume()
```

DESCATALOGADA. La definición de esta llamada depende de `suspend()`, la cual está descatalogada.

run

```
void run()
```

Si este thread fue creado con un objeto de ejecución Runnable por separado, entonces se llama al método `run` de dicho objeto. En otro caso, este método no hace nada y retorna.

setContextClassLoader

```
void setContextClassLoader(ClassLoader cl)
```

Establece el contexto ClassLoader para este thread.

setDaemon

```
void setDaemon(boolean on)
```

Marca este thread como un thread demonio o como un thread de usuario.

setName

```
void setName(String name)
```

Establece el nombre del thread a name.

setPriority

```
void setPriority(int newPriority)
```

Cambia la prioridad de este thread a newPriority.

sleep

```
static void sleep(long millis)
static void sleep(long millis, int nanos)
```

Dependiendo de la variante del método invocado, hace que el thread deje de ejecutarse por millis milisegundos o por millis milisegundos más nanos nanosegundos.

start

```
void start()
```

Hace que el thread empiece a ejecutarse: la Máquina Virtual de Java llama al método `run` de este thread.

stop

```
void stop()
void stop(Throwable obj)
```

DESCATALOGADA. Este método es inherentemente inseguro. Detener un thread con `Thread.stop` provoca desbloquear todos los monitores que haya bloqueado. Si alguno de los objetos que previamente fueron protegidos por estos monitores estuviera en un estado inconsistente, los objetos dañados serían visibles para los demás threads, dando como resultado un comportamiento arbitrario. Muchos usos del método `stop` deberían ser reemplazado por código que modificara alguna variable para indicar que el thread en cuestión debe parar su ejecución. Dicho thread debería consultar regularmente el estado de esa variable para saber cuando debe detener su ejecución. En caso de que los tiempos de espera del thread sean largos, sería conveniente el uso del método `interrupt`.

suspend

```
void suspend()
```

DESCATALOGADA. Si el thread que debe suspender su ejecución tiene en su monitor una reserva sobre un recurso crítico del sistema, cuando suspenda su ejecución ningún thread podrá acceder a dicho recurso hasta que dicho thread vuelva a ejecutarse. Si el thread que debe ejecutar `resume` sobre el thread que posee la reserva del recurso intenta reservar dicho recurso, se produce un deadlock (abrazo mortal)

toString

```
String toString()
```

Devuelve una representación del thread en forma de cadena de caracteres, incluyendo el nombre, prioridad y el grupo del thread.

yield

```
static void yield()
```

Provoca que el thread que se está ejecutando realice una pausa para así permitir que otros threads se ejecuten.

Métodos heredados de la clase `Object`

clone

```
protected Object clone()
```

Crea y devuelve una copia de este objeto.

equals

```
boolean equals(Object obj)
```

Devuelve cierto o falso en función de si este objeto es igual o no al objeto obj.

finalize

```
protected void finalize()
```

Este método es llamado por el Garbage Collector (GC) sobre un objeto cuando el GC determina que no hay más referencias a ese objeto.

getClass

```
Class getClass()
```

Retorna la clase de un objeto en tiempo de ejecución.

hashCode

```
int hashCode()
```

Devuelve un valor de código de hash para este objeto.

notify

```
void notify()
```

Despierta a un único thread que está esperando en el monitor de este objeto.

notifyAll

```
void notifyAll()
```

Despierta a todos los threads que están esperando en el monitor de este objeto.

toString

```
String toString()
```

Devuelve una representación de este objeto en forma de cadena de caracteres.

wait

```
void wait()
void wait(long timeout)
void wait(long timeout, int nanos)
```

Provoca que el thread actual espere hasta que otro thread invoque el método `notify()` o `notifyAll()` para este objeto o, haya transcurrido la cantidad de tiempo indicada por timeout o timeout y nanos.

Referencias y bibliografía

[1] Sun Microsystems, Inc. (1997)
Java Native Interface Specification
Copyright © 1996, 1997 Sun Microsystems, Inc.
http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html
http://java.sun.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.html

[2] Bill Venders (1999)
Inside The Java Virtual Machine, 2nd Edition
McGraw-Hill Professional Publishing, December
ISBN: 0-07-135093-4
http://www.artima.com/insidejvm/blurb.html

[3] Bill Venders (1997)
Under the Hood: How the Java virtual machine performs thread synchronization
Copyright © 1996-2001 Artima
http://www.javaworld.com/javaworld/jw-07-1997/jw-07-hood_p.html
http://www.artima.com/underthehood/thread.html

[4] Doug Lea (1999)
Concurrent Programming in Java™ 2nd Edition: Design Principles and Patterns (The Java™ Series)
Addison-Wesley Pub Co, November
ISBN: 0-20-131009-0
http://gee.cs.oswego.edu/dl/cpj/index.html

[5] Agustín Froufe Quintas (1999)
Java™ 2 Manual de usuario y tutorial
Copyright © 2000 RA-MA
RA-MA Editorial, Diciembre
ISBN: 84-7897-389-3
http://members.es.tripod.de/froufe

[6] Tim Lindholm and Frank Yellin (1999)
The Java™ Virtual Machine Specification, 2nd Edition
Copyright © 1999 Sun Microsystems, Inc.
http://java.sun.com/

[7] Thuan Q. Pham and Pankaj K. Garg (1999)
Multithreaded Programming with Win32
Copyright © 1999 Pretince Hall PTR
ISBN: 0-13-010912-6

[8] Thomas W. Christopher and George K. Thiruvathukal (2001)
High-Performance Java Platform Computing: Multithreaded and Networked Programming
Copyright © 2001 Pretince Hall PTR
Pretince Hall PTR, February
ISBN: 0-13-016164-0

[9] Allen Holub (1998)
Programming Java threads in the real world, Part 1: A Java programmer's guide to threading architecture
JavaWorld, September
http://www.javaworld.com/javaworld/jw-09-1998/jw-09-threads_p.html

[10] Allen Holub (1998)
Programming Java threads in the real world, Part 2: The perils of race conditions, deadlock, and other threading problems
JavaWorld, October
http://www.javaworld.com/javaworld/jw-10-1998/jw-10-toobox_p.html

[11] Bill Venders (1998)
Design for thread safety: Design tips on when and how to use synchronization, immutable objects, and thread-safe wrappers
JavaWorld, August
http://www.javaworld.com/javaworld/jw-08-1998/jw-08-techniques_p.html

[12] Peter A. Buhr, Michael Fortier, and Michael H. Coffin (1995)
Monitor Classification
ACM Computing Surveys, v.27 n.1, p.63-107, March
ftp://plg.uwaterloo.ca/pub/uSystem/MonitorClassification.ps.gz

[13] Gregory R. Andrews, and Fred B. Schneider (1983)
Concepts and Notations for Concurrent Programming
ACM Computing Surveys, v.15 n.1, p.3-43, March
http://www.cs.cornell.edu/fbs/publications/LangSurv.pdf

[14] Anita J. Van Engen, Michael K. Bradshaw, and Nathan Oostendorp (2001)
Extending Java to support shared resource protection and deadlock detection in threads programming
Copyright © 2000-2002 ACM, Inc., January
http://www.acm.org/crossroads/xrds4-2/dynac.html

[15] Per Brinch Hansen (1999)
Java's insecure parallelism
ACM SIGPLAN Notices, v.34 n.4, p.38-45, April

[16] David Mosberger and Stéphane Eranian (2001)
ia-64 Linux® kernel design and implementation
Copyright © 2002 Hewlett-Packard Company
Pretince Hall PTR, November 2001
ISBN: 0-13-061014-3
http://www.lia64.org/book/

[17] José Oliver, Eduard Ayguadé, Nacho Navarro (2000)
Towards an efficient explotation of loop-level parallelism in Java
Technical Report UPC-DAC-2000-24, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya
http://www.ac.upc.es/recerca/reports/DAC/2000/index,en.html#UPC-DAC-2000-24

[18] Albert Serra, Marisa Gil, Nacho Navarro (1996)
Planificació orientada a les aplicacions paral.leles en multiprocessadors multiprogramats de memoria compartida
Technical Report UPC-DAC-1996-54, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya

http://www.ac.upc.es/recerca/reports/DAC/1996/index,en.html#UPC-DAC-1996-54

[19] Sun Microsystems, Inc. (1996)
JDK 1.1 for Solaris Developer's Guide
Copyright © 1996, 1999 Sun Microsystems, Inc.
Part Number 806-3461-10
http://docs.sun.com/?p=/doc/806-3461

[20] Douglas Kramer, Bill Joy and David Spenhoff (1996)
The Java™ Platform (a white paper)
Copyright © 1996, 1999 Sun Microsystems, Inc., May
http://java.sun.com/docs/white/index.html

[21] Bruce McCormick (January 1999)
Three POSIX Threads' implementations
http://www.nswc.navy.mil/cosip/feb99/cots0299-1.shtml
http://www.nswc.navy.mil/cosip/index.html

[22] Sun Microsystems, Inc. (2000)
Multithreaded Programming Guide
Copyright © 2000 Sun Microsystems, Inc.
Part Number 806-1388-10
http://docs.sun.com/?p=/doc/806-6867

[23] International Business Machines Corporation (1999)
AIX Version 4.3 General Programming Concepts: Writing and Debugging Programs
Second Edition, September
Copyright © 1997, 1999 International Business Machines Corporation.
http://publib.boulder.ibm.com/doc_link/en_US/a_doc_lib/aixprggd/genprogc/toc.htm
http://publib.boulder.ibm.com/doc_link/en_US/a_doc_lib/aixprggd/genprogc/understanding_threads.htm

[24] Mark Secrist and Laura Smyrl (2000)
Threads and the Java™ virtual machine
AppDev ATC, December.
Copyright © 1994, 2002 Hewlett-Packard Company.
http://h21007.www2.hp.com/dspp/tech/tech_TechDocumentDetailPage_IDX/1,1701,390,00.html

[25] Bryan M. Cantrill and Thomas W. Doeppner Jr. (1997)
ThreadMon: A Tool for Monitoring Multithreaded Program Performance
RI 02912-1910, Department of Computer Sciencie, Brown University
http://www.cs.brown.edu/research/thmon/thmon.html

[26] Bil Lewis and Daniel J. Berg (1998)
Multithreaded Programming with Pthreads
Copyright © 1998 Sun Microsystems, Inc.
Pretince Hall PTR
ISBN: 0-13-680729-1

Índice alfabético

T

W