13. Gestión de hilos de ejecución.

En este capítulo se mostrará una de las características relevantes que caracterizan a las aplicaciones Java, la concurrencia de tareas. Como bien es sabido, una de las características que hacen que Java sea empleado en servidores de aplicaciones con grandes volúmenes de peticiones es la multitarea. La multitarea en Java se logra a traves de los hilos.

Hasta ahora se ha visto en capítulos anteriores como la maquina virtual ejecuta un programa sencillo, como gestiona la memoria de los objetos que se crean y como se ejecutan los métodos. En esta capítulo, se abordara en detalle como se comporta internamente la maquina virtual cuando se han de ejecutar aplicaciones multihilo.

Previamente al estudio de la implementación interna de la ejecución multihilo en la KVM, se mostrará un apartado introductorio de esta potente herramienta. Partiendo del concepto de multitarea se llegará hasta las distintas formas de implementar esta multitarea.

Ya sumergiéndonos en el ámbito de la KVM, la gestión multihilo se encuadra dentro del tipo combinación ULT-KLT que como se verá en la introducción combina gestión de los hilos a nivel software (por parte de la maquina virtual) y a nivel sistema operativo (tareas que realiza el sistema operativo). En particular el algoritmo que se emplea en la planificación de hilos dentro de la KVM es conocido como round-robin y es estudiado con detalle en este capítulo.

Un aspecto crucial cuando se trata con sistemas multihilo es la comunicación entre los hilos. Java permite que distintos hilos puedan acceder a una misma instancia de un objeto o elementos del mismo, es lo que se conoce como sincronización entre hilos. Es por ello que la KVM dispone de un mecanismo de sincronismo de objetos compartidos transparente al usuario para controlar el acceso simultáneo a dichos objetos. Básicamente este mecanismo consiste en la implementación de un semáforo en tres niveles para poder acceder desde un hilo a cualquier objeto.

13.1. Introducción.

Un **hilo de ejecución**, en los sistemas operativos, es similar a un proceso en que ambos representan una secuencia simple de instrucciones ejecutada en paralelo con otras secuencias. Los hilos permiten dividir un programa en dos o más tareas que corren simultáneamente, por medio de la multiprogramación. En realidad, este método permite incrementar el rendimiento de un procesador de manera considerable. En todos los sistemas de hoy en día los hilos son utilizados para simplificar la estructura de un programa que lleva a cabo diferentes funciones.

Todos los hilos de un proceso comparten los recursos del proceso. Residen en el mismo espacio de direcciones y tienen acceso a los mismos datos. Cuando un hilo modifica un dato en la memoria, los otros hilos utilizan el resultado cuando acceden al dato. Cada hilo tiene su propio estado, su propio contador, su propia pila y su propia

copia de los registros de la CPU. Los valores comunes se guardan en el bloque de control de proceso (PCB), y los valores propios en el bloque de control de hilo (TCB). Muchos lenguaje de programación (como Java), y otros entornos de desarrollo soportan los llamados **hilos** o **hebras** (en inglés, *threads*).

Un ejemplo de la utilización de hilos es tener un hilo atento a la interfaz gráfica (iconos, botones, ventanas), mientras otro hilo hace una larga operación internamente. De esta manera el programa responde más ágilmente a la interacción con el usuario.

13.1.1. Diferencias entre hilos y procesos.

Los hilos se distinguen de los tradicionales procesos en que los procesos son generalmente independientes, llevan bastante información de estados, e interactúan sólo a través de mecanismos de comunicación dados por el sistema. Por otra parte, muchos hilos generalmente comparten otros recursos directamente. En los sistemas operativos que proveen facilidades para los hilos, es más rápido cambiar de un hilo a otro dentro del mismo proceso, que cambiar de un proceso a otro. Este fenómeno se debe a que los hilos comparten datos y espacios de direcciones, mientras que los procesos al ser independientes no lo hacen. Al cambiar de un proceso a otro el sistema operativo (mediante el *dispacher*) genera lo que se conoce como *overhead*, que es tiempo desperdiciado por el procesador para realizar un cambio de modo (*mode switch*), en este caso pasar del estado de *Runnig* al estado de *Waiting* o Bloqueado y colocar el nuevo proceso en *Running*. En los hilos como pertenecen a un mismo proceso al realizar un cambio de hilo este *overhead* es casi despreciable.

Sistemas operativos como Windows NT, OS/2 y Linux (2.5 o superiores) han dicho tener hilos "baratos", y procesos "costosos" mientras que en otros sistemas no hay una gran diferencia.

13.1.2. Funcionalidad de los hilos.

Al igual que los procesos, los hilos poseen un estado de ejecución y pueden sincronizarse entre ellos para evitar problemas de compartimiento de recursos. Generalmente, cada hilo tiene especificada una tarea específica y determinada, como forma de aumentar la eficiencia del uso del procesador.

Los principales estados de ejecución de los hilos son: Ejecución, Listo y Bloqueado. No tiene sentido asociar estados de suspensión de hilos ya que es un concepto de proceso. En todo caso, si un proceso está expulsado de la memoria principal (ram), todos sus hilos deberán estarlo ya que todos comparten el espacio de direcciones del proceso. Las transiciones entre estados más comunes son las siguientes:

- *Creación:* Cuando se crea un proceso se crea un hilo para ese proceso. Luego, este hilo puede crear otros hilos dentro del mismo proceso. El hilo tendrá su propio contexto y su propio espacio de pila, y pasara a la cola de listos.

- *Bloqueo:* Cuando un hilo necesita esperar por un suceso, se bloquea (salvando sus registros). Ahora el procesador podrá pasar a ejecutar otro hilo que este en la cola de Listos mientras el anterior permanece bloqueado.

- *Desbloqueo:* Cuando el suceso por el que el hilo se bloqueo se produce, el mismo pasa a la cola de Listos.

- *Terminación:* Cuando un hilo finaliza se liberan tanto su contexto como sus pilas.

13.1.3. Ventajas de los hilos frente a los procesos.

Si bien los hilos son creados a partir de la creación de un proceso, podemos decir que un proceso es un hilo de ejecución, conocido como *monohilo*. Pero las ventajas de los hilos se dan cuando hablamos de *Multihilos*, un procesos tiene múltiples hilos de ejecución los cuales realizan actividades distintas, que pueden o no ser cooperativas entre si. Los beneficios de los hilos se derivan de las implicaciones de rendimiento y se pueden resumir en los siguientes puntos:

- Se tarda mucho menos tiempo en crear un hilo nuevo en un proceso existente que en crear un proceso. Algunas investigaciones llevan al resultado que esto es así en un factor de 10.

- Se tarda mucho menos en terminar un hilo que un proceso, ya que su cuando se elimina un proceso se debe eliminar el PCB del mismo, mientras que un hilo se elimina su contexto y pila.

- Se tarda mucho menos tiempo en cambiar entre dos hilos de un mismo proceso.

- Los hilos aumentan la eficiencia de la comunicación entre programas en ejecución. En la mayoría de los sistemas en la comunicación entre procesos debe intervenir el núcleo para ofrecer protección de los recursos y realizar la comunicación misma. En cambio, entre hilos pueden comunicarse entre si sin la invocación al núcleo. Por lo tanto, si hay una aplicación que debe implementarse como un conjunto de unidades de ejecución relacionadas, es más eficiente hacerlo con una colección de hilos que con una colección de procesos separados.

13.2. Aspectos íntimos de la ejecución multihilo.

Hay una serie de aspectos a la hora de implementar un sistema de gestión de hilos muy críticos como puede ser la sincronización entre hilos y que en algunos casos son dependientes del hardware que se este usando o bien de la plataforma de desarrollo (que versión de la maquina virtual Java se esta usando por ejemplo) o incluso del propio código que los desarrolladores de software crean.

13.2.1. Sincronización entre hilos.

Todos los hilos comparten el mismo espacio de direcciones y otros recursos como pueden ser archivos abiertos. Cualquier modificación de un recurso desde un hilo afecta al entorno del resto de los hilos del mismo proceso. Por lo tanto, es necesario sincronizar la actividad de los distintos hilos para que no interfieran unos con otros o corrompan estructuras de datos.

Una ventaja de la programación multihilo es que los programas operan con mayor velocidad en sistemas de computadores con múltiples CPUs (sistemas multiprocesador o a través del grupo de máquinas) ya que los hilos del programa se prestan verdaderamente para la ejecución concurrente. En tal caso el programador necesita ser cuidadoso para evitar condiciones de carrera (problema que sucede cuando diferentes hilos o procesos alteran datos que otros también están usando), y otros comportamientos no intuitivos. Los hilos generalmente requieren reunirse para procesar los datos en el orden correcto. Es posible que los hilos requieran de *operaciones atómicas* para impedir que los datos comunes sean cambiados o leídos mientras estén siendo modificados, para lo que usualmente se utilizan los semáforos. El descuido de esto puede generar interbloqueo.

13.2.2. Formas de los multihilos.

Los Sistemas Operativos generalmente implementan hilos de dos maneras:

- *Multihilo apropiativo:* Permite al sistema operativo determinar cuándo debe haber un *cambio de contexto*. La desventaja de esto es que el sistema puede hacer un cambio de contexto en un momento inadecuado, causando un fenómeno conocido como inversión de prioridades y otros problemas.
- *Multihilo cooperativo:* Depende del mismo hilo abandonar el control cuando llega a un punto de detención, lo cual puede traer problemas cuando el hilo espera la disponibilidad de un recurso.

El soporte de *hardware* para *multihilo* desde hace poco se encuentra disponible. Esta característica fue introducida por Intel en el Pentium 4, bajo el nombre de HyperThreading.

13.2.3. Usos más comunes.

Algunos de los casos más claros en los cuales el uso de hilos se convierte en una tarea fundamental para el desarrollo de aplicaciones son:

- Trabajo interactivo y en segundo plano: por ejemplo, en un programa de hoja de cálculo un hilo puede estar visualizando los menús y leer la entrada del usuario mientras que otro hilo ejecuta las órdenes y actualiza la hoja de cálculo. Esta medida suele aumentar la velocidad que se percibe en la aplicación, permitiendo que el programa pida la orden siguiente antes de terminar la anterior.

- Procesamiento asíncrono: los elementos asíncronos de un programa se pueden implementar como hilos. Un ejemplo es como los software de procesamiento de texto guardan archivos temporales cuando se esta trabajando en dicho programa. Se crea un hilo que tiene como función guardar una copia de respaldo mientras se continúa con la operación de escritura por el usuario sin interferir en la misma.

- Aceleración de la ejecución: se pueden ejecutar, por ejemplo, un lote mientras otro hilo lee el lote siguiente de un dispositivo.

- Estructuración modular de los programas: puede ser un mecanismo eficiente para un programa que ejecuta una gran variedad de actividades, teniendo las mismas bien separadas mediante a hilos que realizan cada una de ellas.

13.3. Implementaciones.

Hay dos grandes categorías en la implementación de hilos:

- Hilos a nivel de usuario
- Hilos a nivel de Kernel

También conocidos como **ULT** (*User Level Thread*) y **KLT** (*Kernel Level Thread*).

13.3.1. Hilos a nivel de usuario (ULT).

En una aplicación *ULT* pura, todo el trabajo de gestión de hilos lo realiza la aplicación y el núcleo o *kernel* no es consciente de la existencia de hilos. Es posible programar una aplicación como *multihilo* mediante una biblioteca de hilos. La misma contiene el código para crear y destruir hilos, intercambiar mensajes y datos entre hilos, para planificar la ejecución de hilos y para salvar y restaurar el contexto de los hilos. Todas las operaciones descritas se llevan a cabo en el espacio de usuario de un mismo proceso. El *kernel* continua planificando el proceso como una unidad y asignándole un único estado (Listo, bloqueado, etc.)

Las ventajas que ofrecen los ULT son:

- El intercambio de los hilos no necesita los privilegios del modo kernel, por que todas las estructuras de datos están en el espacio de direcciones de usuario de un mismo proceso. Por lo tanto, el proceso no debe cambiar a modo kernel para gestionar hilos. Se evita la sobrecarga de cambio de modo y con esto el sobrecoste u overhead.

- Se puede realizar una planificación específica. Dependiendo de que aplicación sea, se puede decidir por una u otra planificación según sus ventajas.

- Los ULT pueden ejecutar en cualquier sistema operativo. La biblioteca de hilos es un conjunto compartido.

Igualmente las desventajas más relevantes son:

- En la mayoría de los sistemas operativos las llamadas al sistema (*System calls*) son bloqueantes. Cuando un hilo realiza una llamada al sistema, se bloquea el mismo y también el resto de los hilos del proceso.

- En una estrategia ULT pura, una aplicación multihilo no puede aprovechar las ventajas de los multiprocesadores. El núcleo asigna un solo proceso a un solo procesador, ya que como el núcleo no interviene y ve al conjunto de hilos como un solo proceso.

Una solución al bloqueo mediante a llamadas al sistema es usando la técnica de ***jacketing***, que es convertir una llamada bloqueante en no bloqueante.

Algunos ejemplos de sistemas ULT son: GNU Portable Threads, FSU Threads, Apple Computer Thread Manager, REALbasic´s cooperative threads.

13.3.2. Hilos a nivel de Kernel de sistema operativo (KLT).

En una aplicación KLT pura, todo el trabajo de gestión de hilos lo realiza el *kernel*. En el área de la aplicación no hay código de gestión de hilos, únicamente un API (interfaz de programas de aplicación) para la gestión de hilos en el núcleo. Windows 2000, Linux y OS/2 utilizan este método.

Las principales ventajas de los KLT:

- El kernel puede planificar simultáneamente múltiples hilos del mismo proceso en múltiples procesadores.

- Si se bloquea un hilo, puede planificar otro del mismo proceso.

- Las propias funciones del kernel pueden ser multihilo

En cambio las desventajas que presentan son:

- El paso de control de un hilo a otro precisa de un cambio de modo.

Algunos ejemplos de sistema KLT son: Light Weight Kernel Threads, M:N threading, Native POSIX Thread Library para Linux.

13.3.3. Combinaciones ULT-KLT.

Algunos sistemas operativos ofrecen la combinación de ULT y KLT, como Solaris. La creación de hilos, así como la mayor parte de la planificación y sincronización de los hilos de una aplicación se realiza por completo en el espacio de

usuario. Los múltiples ULT de una sola aplicación se asocian con varios KLT. El programador puede ajustar el número de KLT para cada aplicación y máquina para obtener el mejor resultado global.

Si bien no hay muchos ejemplos de sistemas de este tipo si se pueden encontrar implementaciones híbridas auxiliares en algunos sistemas como por ejemplo las planificaciones de activaciones de hilos que emplea la librería NetBSD native POSIX threads. Esta implementación no es más que un modelo N:M que se contrapone al modelo 1:1 de que dispone el Kernel o el espacio de desarrollo del usuario.

13.4. Funcionamiento de la maquina virtual.

Vamos a comentar a continuación de forma breve cual es el funcionamiento de la maquina virtual si bien dicha explicación se puede encontrar con un mayor nivel de detalle en capítulo *5 Modelo de funcionamiento de la maquina virtual*. Esta explicación tiene como objeto centrarnos en el contexto en el cual vamos a movernos.

La estructura de la maquina virtual es la siguiente:

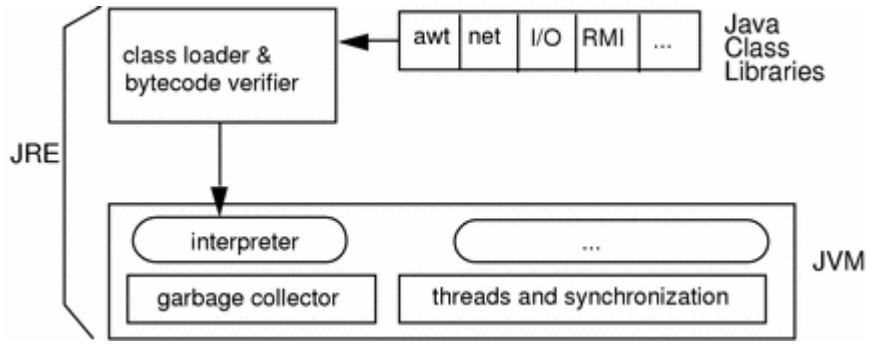

Figura 13.1: Modelo funcionamiento KVM.

A partir de los archivos *.class* ya compilados el cargador de clases (*loader*) es encarga como su propio nombre indica de cargar las clases en la maquina virtual reservando para ello la memoria necesaria para los datos en las áreas destinadas a tal efecto (*runtime data areas*). Cada hilo creado durante la ejecución de la maquina virtual comparte este área con el resto de los hilos pero cada hilo tiene sus propios registros de funcionamiento. Es decir cada hilo tiene su propio registro IP o contador de instrucciones que indica cual es la instrucción a nivel de *bytecodes* que debe ser ejecutada y por supuesto su propia pila de ejecución (accesible por el registro SP).

La pila de ejecución del hilo mantiene la información de ejecución de dicho hilo tal como los métodos que se han de ejecutar junto son sus parámetros y sus valores de retorno si lo tienen, parámetros relacionados con el sincronismo entre hilos, etc.…

Precisamente existen una serie de registros que son empleados por el interprete para poder completar el flujo de *bytecodes* correspondientes a un hilo (.class):

- IP (instruction pointer): es el puntero que indica la instrucción que se esta ejecutando.

- SP (stack pointer): puntero a la pila de ejecución del hilo.
- LP (locals pointers): punteros locales necesarios para parámetros auxiliares del hilo.
- FR (frame pointer): puntero a la lista de marcos de ejecución de los métodos del hilo.
- CP (constant pool pointer): puntero a la constante de almacenaje de datos del hilo.

Estos registros al ser propios de cada hilo Java que se ejecuta en la maquina virtual. Por ello en aplicaciones multihilo cada vez que un hilo que se estaba ejecutando pasa a estar suspendido debe mantener una copia del valor de los registros antes de ser suspendido para volver a cárgalos en la maquina virtual cuando vuelva a ejecutarse.

13.5. Implementación del módulo de gestión de hilos de la KVM.

A continuación vamos a documentar de forma exhaustiva como funciona el módulo de gestión de hilos de la KVM. Este módulo es del tipo KLT, es decir Java en realidad lo que hace es proporcionar una interfaz a través de la cual el programador puede acceder a las funciones nativas del kernel del sistema que realmente si gestión los hilos que puedan estar ejecutándose.

Este módulo es posiblemente uno de los que mas modificaciones ha tenido desde la primera versión de la KVM. El motivo de ello es que la gestión de hilos en un sistema es un aspecto muy complejo y que afecta mucho al rendimiento global del sistema. Es por ello que un mal uso de este módulo o bien una deficiencia en la implementación del mismo redunda en problemas diversos: aumento de la carga de CPU, tiempos de espera muy altos, etc.. Como ya se comento en los primeros capítulos estos son precisamente los problemas que si bien pueden permitirse en sistemas informáticos de gran capacidad, son absolutamente inadmisibles en sistemas móviles y de baja capacidad.

El modelo de planificación de hilos que emplea la KVM tiene una serie de características que lo hacen muy interesante:

- Independiente del hardware.
- Portable.
- Con derecho preferente.
- Independiente del lenguaje Java.

Básicamente la KVM emplea un algoritmo de planificación denominado *round-robin* en el cual todos los hilos activos en el sistema están almacenados en una lista circular esperando a ser ejecutados en el preciso momento.

El algoritmo round-robin es una los algoritmos de planificación de hilos mas sencillos que existen (de ahí su elección para la KVM). Básicamente este algoritmo asigna a cada hilo porciones de tiempo similares y en orden va ejecutando cada uno de los hilos sin ningún tipo de prioridad entre ellos. Este algoritmo es muy simple tanto de

ejecutar como de implementar y además evita una reserva infinita de recursos en caso de que se produzcan situaciones anómalas de ejecución (por ejemplo que un hilo deje bloqueado un archivo aun cuando este hilo haya dejado de usarlo).

De hecho como reseña cabe decir que este algoritmo se emplea no sea para ejecución multitarea sino también por ejemplo para planificación en las redes para el acceso de los distintos equipos a los recursos compartidos por toda la red. La forma de trabajar del algoritmo round-robin sería la siguiente:

Supongamos que la unidad de tiempo asignada a cada hilo es de 100ms y tenemos una primera tarea que tarda 250ms en ser completada. El planificador de hilos tras haber ejecutado 100ms de la primera tarea pasa a ejecutar otra tarea dejando la primera suspendida. Una vez que el resto de tareas han sido ejecutadas hasta su primera porción de tiempo volvería a ejecutar otros 100ms de la primera tarea y se suspendería al acabar su tiempo de uso de procesador. Ya en una tercera pasada la primera tarea solo usaría 50 ms de proceso que es lo que faltaba para terminar su ejecución.

En el caso de la KVM las tareas son hilos de ejecución que son almacenados en una lista circular para ir siendo accedidos de acuerdo con el algoritmo uno detrás de otro pero haciendo uso de las prioridades que desde Java se han asignado a cada hilo (lo cual es una variante del algoritmo round-robin). Estas prioridades como ya se sabe no son mas que valores enteros que indican al interprete de *bytecodes* de la maquina virtual cuantos hilos han de ejecutarse anteriormente. Por cada *bytecode* ejecutado por la KVM el intérprete decrementa el contador *TimeSlice* que actúa como temporizador de ejecución del hilo. De forma que cuando este contador llegue a cero el hilo que estaba ejecutando se suspende y se inicia la ejecución de otro hilo (*thread switching*).

Este modelo de hilos de ejecución descrito en el apartado anterior ha sufrido algunas modificaciones en la versión 1.0 y posteriores de la KVM y se pueden resumir como sigue:

- Soporte opcional para permitir operaciones de entrada/salida (como escritura en un fichero) sin que por ello bloquee el mecanismo de hilos de ejecución. Otra de las modificaciones introducidas fue la reestructuración del código con vistas a que facilitar las modificaciones del mecanismo de planificación de hilos sin que ello conlleve un cambio total del sistema.

- Hasta la versión 1.0 de la KVM la operación de *thread switching* (suspender un hilo para ejecutar otro) se realizaba en múltiples puntos de ejecución de la maquina virtual. A partir de la versión 1.0 dicha operación solo se ejecuta en un único punto que es al principio de la ejecución de cada uno de los *bytecodes* en el intérprete.

- En la versión antigua los hilos que habían de ejecutarse (runnable threads) se almacenaban en una lista circular como ya hemos comentado con anterioridad incluyendo el hilo que se esta ejecutando en ese momento. En la versión actual de la KVM cada hilo se elimina de la lista circular cuando se comienza su ejecución y se vuelve a introducir en la lista cuando el hilo es bloqueado sea por el motivo que sea.

- Se añadió una variable global accesible desde cualquier punto de la maquina virtual *CurrentThread* que referencia al hilo que se esta ejecutando en ese momento. Y una segunda variable *RunnableThreads* que referencia a la lista de circular de hilos pendientes de ejecutarse.

- Además se ha incorporado cambios importantes en muchas de las rutinas del módulo de gestión de hilos y que se irán comentado mas adelante conforme vayamos viendo en detalle la implementación de Sun para las operaciones de este módulo.

- La versión 1.0.3 además dispone de una implementación para el sincronismo y comunicación entre hilos reformado y que se comenta en detalle en apartados posteriores de este mismo capítulo.

Por otro lado el lenguaje Java gestiona y maneja los hilos desde el punto de vista del usuario tal y como se indica en la figura siguiente:

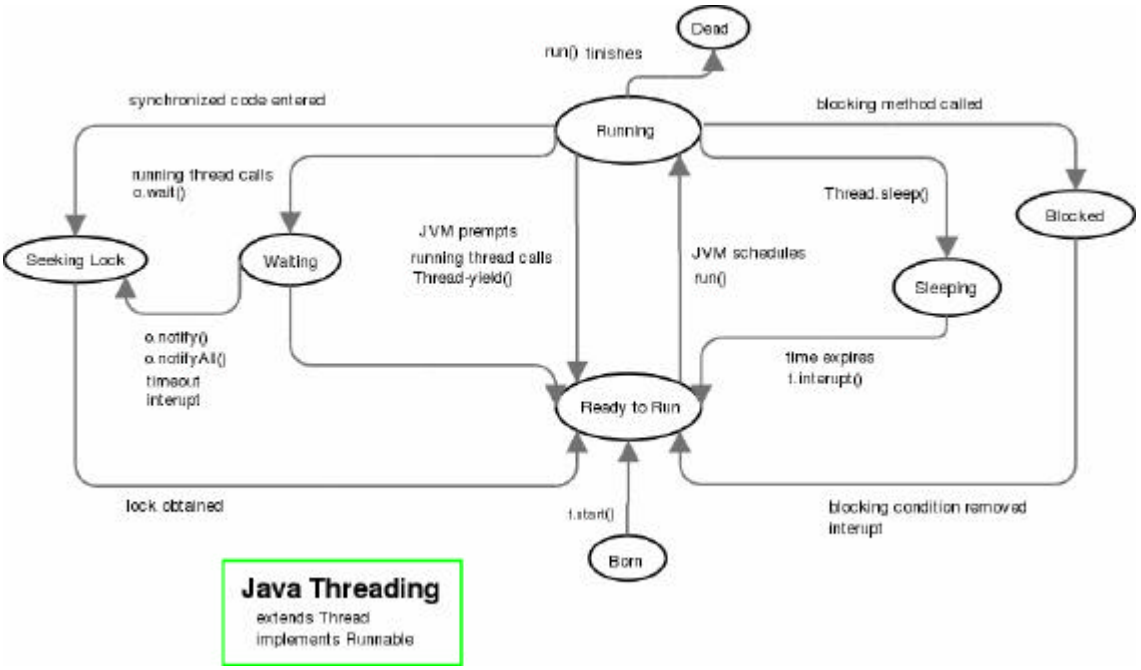

Figura 13.2: Modelo gestión multihilo Java.

Para obtener una descripción mas detallada acerca de la API de gestión de hilos de Java se recomienda la lectura de la bibliografía asociada a tal tema.

13.6. Estructuras de datos para representación de hilos de ejecución.

Justo al comienzo del fichero thread.h que contiene las declaraciones de variables y métodos que se emplean en el fichero thread.c nos encontramos con las siguientes variables:

```
extern THREAD CurrentThread;    /* Current thread */
```

```
extern THREAD MainThread;       /* For debugger code to access */

extern THREAD AllThreads;       /* List of all threads */
extern THREAD RunnableThreads;   /* Queue of all threads that can be
run */

extern int AliveThreadCount;    /* Number of alive threads */

extern int Timeslice;           /* Time slice counter for multitasking
*/


#define MAX_PRIORITY  10        /* These constants must be the same */
#define NORM_PRIORITY 5         /* as those defined in Thread.java */
#define MIN_PRIORITY  1
```

Que como sabemos representan el hilo que se esta ejecutando en cada momento (*CurrentThread*), la lista de todos los hilos que existen en el sistema (*AllThread*), el hilo principal en el cual se crean el resto de hilos (*MainThread*), el número de hilos activos (*AliveThreadCounts*) así como el rango de prioridades (que debe coincidir con el definido en la clase *Thread* de Java).

De entre estas variables los hilos o listas de los mismos se representan mediante el tipo THREAD definido este como un puntero a una lista de objetos *threadQueue* :

```
typedef struct threadQueue*         THREAD;
```

Llegado a este punto la estructura *threadQueue* es la que representa a un hilo en particular. Esta estructura esta integrada por lo siguientes elementos:

- *nextAliveThread*: puntero al siguiente hilo activo (para implementar la lista de hilos activos anteriormente comentada).
- *nextThread*: puntero al siguiente hilo (para implementar la lista de hilos totales presentes en el sistema). Tener en cuenta que estos dos punteros son nulos en el caso de que se este representando un único hilo como es el caso de *CurrentThread*.
- *javaThread*: puntero a una variable del tipo JAVATHREAD que sería una instancia Java del hilo en cuestión. Un poco más adelante aclararemos el porque de esta distinción dentro del hilo.
- *Timeslice*: intervalo de tiempo asignado a los hilos para que estos se ejecuten de manera continuada y sin interrupciones.
- *Stack*: puntero a un objeto del tipo STACK, es decir la pila de ejecución del hilo.
- Variables que se emplean para almacenar los valores de los registros de la maquina virtual del hilo mientras el hilo se encuentra en estado activo a la espera de ser ejecutado:

```
    BYTE* ipStore;          /* Contador de rogramas */
    FRAME fpStore;          /* Puntero temporal al marco de
ejecución del hilo*/
    cell* spStore;          /* Puntero a pila de ejecución
temporal*/
  cell* nativeLp;         /* Usado en la interfaz KNI para acceso a
variables locales*/
```

De esta forma cuando el hilo pase a ejecutarse el intérprete puede recuperar fácilmente los valores de los registros para seguir ejecutando los *bytecodes* de dicho hilo.

- *monitor*: variable del tipo MONITOR que hace referencia a la cola de espera en la cual se encuentra el hilo para ejecutar un método sincronizado.
- *monitor_depth*: número de elementos en el monitor en concreto.
- *nextAlarmThread*: simplemente es un puntero a la lista de hilos que deben ser avisados cuando se sale de un método sincronizado.
- *weakUpCall* : función de callback que se emplea como sustitución de la alarma anterior (opción configurable):

```
void (*wakeupCall)(THREAD);
```

- *extendedLock*: mecanismo de bloqueo de recursos especial usado solo por objetos que soporten FASTLOCK:

- *pendingException* y *exceptionMessage*: representan la clase de la excepción asociada a una función nativa que aún no ha sido capturada y el mensaje asociado a la misma. Estas variables se emplean para controlar excepciones de sistema no definidas en Java:

```
#if (ASYNCHRONOUS_NATIVE_FUNCTIONS || USE_KNI)
      char* pendingException;
      char* exceptionMessage;
#endif
```

- *state*: listado de los estados en los que puede encontrarse el hilo:

```
enum {
  THREAD_JUST_BORN = 1,     //CREADO PERO NO INCIADO
  THREAD_ACTIVE = 2,        //INCIADO PERO NO EN EJECUCION
  THREAD_SUSPENDED = 4,     //SUSPENDIDO
  THREAD_DEAD = 8,          //ELIMINADO
  THREAD_MONITOR_WAIT = 16, //ESPERANDO ACCESO SINCRONIZADO
  THREAD_CONVAR_WAIT = 32,
  THREAD_DBG_SUSPENDED = 64  //SUSPENDIDO DESDE EL DEPURADOR
    } state;
```

- Variables varias empleadas solo en depuración tales como saber si el hilo esta ejecutándose (*isStepping*), esta detenido en un punto de ruptura (*isAtBreakPoint*), esta esperando un evento (*needEvent*) y otras.

```
#if ENABLE_JAVA_DEBUGGER
    bool_t isStepping;
    bool_t isAtBreakpoint;
    bool_t needEvent;
    CEModPtr debugEvent;

    ByteCode nextOpcode;
    struct singleStep_mod stepInfo;
    int debugSuspendCount;
      #endif
```

Hemos visto como dentro de la estructura que se emplea para representar el hilo de ejecución solo existe una única referencia al propio código Java del hilo y es a través de puntero JAVATHREAD. De esta forma THREAD es una estructura interna que emplea la maquina virtual para la planificación de hilos independiente de la implementación Java del hilo en sí. Así JAVATHREAD mantiene una relación uno a uno con la estructura definida en la clase Java Thread.java.

Entonces ¿Por qué es necesaria esta separación?. Los hilos en Java son en realidad subclases de la clase Thread por lo cual no podemos añadir los distintos campos que integran la clase directamente al objeto THREAD y es por ello que se emplea una estructura independiente en su lugar. Además esta separación aumenta aún más si cabe la portabilidad de este módulo de gestión de hilos:

```
  struct javaThreadStruct {

COMMON_OBJECT_INFO(INSTANCE_CLASS)
long      priority;
THREAD    VMthread;
INSTANCE  target;
  };
```

Así JavaThread mantiene en su interior varios elementos:

- *COMMON_OBJECT_INFO(INSTANCE_CLASS)*: puntero a la clase del objeto que representa el hilo.
- *Priority*: prioridad actual del hilo.
- *VMThread*: puntero al hilo THREAD al cual pertenece este hilo JAVA.
- *Target*: puntero la instancia del objeto en cuestión (INSTANCE) ya que como hemos dicho el hilo no es más que un objeto Java como otro cualquiera y como tal es ejecutado

13.7. Implementación operaciones globales módulo planificación de hilos.

Este módulo como ya hemos aplicado es accesible actualmente desde el intérprete de *bytecodes* que es el encargado en invocar al planificador de hilos después de cada ejecución correcta o incorrecta de un *bytecode*. Básicamente existen tres operaciones por las cuales el resto de módulos de la maquina virtual pueden invocar al planificador y son:

```
void storeExecutionEnvironment(THREAD thisThread);
void loadExecutionEnvironment(THREAD thisThread);
bool_t SwitchThread(void);
```

Veamos a continuación cada una de estas operaciones.

13.7.1. Salvaguarda y recuperación del entorno de ejecución.

Mediante los métodos *storeExecutionEnvironment* y *loadExecutionEnvironment* el intérprete puede salvaguardar los registros de la maquina virtual y cargarlos respectivamente. De esta forma la función *storeExecutionEnvironment* almacena en las variables del hilo que le es pasado como parámetros los valores de los registros de la maquina virtual en ese momento:

```
thisThread->fpStore = getFP();
thisThread->spStore = getSP();
thisThread->ipStore = getIP();
```

De similar forma el método *loadExecutionEnvironment* carga en los registros del sistema las copias de dichos registros del hilo que le es pasado como parámetro:

```
setFP(thisThread->fpStore);
setLP(FRAMELOCALS(getFP()));
setCP(getFP()->thisMethod->ofClass->constPool);
setSP(thisThread->spStore);
setIP(thisThread->ipStore);
```

Para esa carga se usan las macros setXX donde XX es el registro en cuestión y que están definidas en el intérprete (capítulo 9).

13.7.2. Operación *thread switching*.

Esta es la operación que el intérprete ejecuta para suspender el hilo que se estaba ejecutando y pasar a ejecutar el hilo con mayor prioridad que esta esperando a ser ejecutado. Esta es la operación principal de este módulo y devuelve un valor lógico si se tras ser invocado indicando si existe algún hilo ejecutándose.

En la invocación a esta operación el hilo activo *CurrentThread* se puede encontrar en tres estados diferentes:

- No existe ningún hilo ejecutándose en el sistema debido a que el método *suspendThread*() para dicho hilo ha sido invocado (*CurrentThread* == NULL).
- Existe un hilo ejecutándose y su estado es activo lo cual sería el caso normal que se da cuando el *timeslice* del hilo llega a cero.
- Existe un hilo ejecutándose pero su estado no es activo lo cual es una situación bastante atípica que se produce cuando se llama a la operación *suspendThread*() en modo supervisor por ejemplo cuando se esta depurando un programa *multihilo* Java.

Veamos en detalle cada una de las fases de ejecución de esta operación. Se pueden distinguir fases bien definidas:

- Comprobación estado del hilo ejecutándose actualmente.
- Buscar hilo nuevo a ejecutarse.
- Añadir el hilo antiguo la lista donde se almacenan el resto de hilos esperando a ser ejecutados.
- Restauran los registros de la maquina virtual con los almacenados en el hilo que se ejecuta ahora.
- Actualizan estadísticas.

Como hemos comentado se comprueba en primer lugar si ya existe un hilo ejecutándose. Si no fuera así directamente se recurriría a la fase 2. Si por el contrario existe un hilo ejecutándose se comprueba si el hilo que se esta ejecutando en ese momento mantiene alguna excepción nativa que aun no ha lanzado en cuyo caso se genera un error fatal indicando al usuario de esta situación:

```
#if ASYNCHRONOUS_NATIVE_FUNCTIONS
  if (CurrentThread->pendingException != NIL) {
      fatalError(KVM_MSG_BAD_PENDING_EXCEPTION);
  }
#endif
```

Posteriormente se comprueba si el hilo que se esta ejecutando en ese momento esta activo, en caso de que no lo fuera se genera el correspondiente error fatal al usuario (esta situación se da en raros casos):

```
if (CurrentThread->state == THREAD_ACTIVE) {

        } else {

fatalError(KVM_MSG_ATTEMPTING_TO_SWITCH_TO_INACTIVE_THREAD);
        }
```

Si el hilo si esta activo que será la situación más típica se ha de preparar el hilo para el intercambio. Así si no hay ningún hilo en la cola *RunnableThreads*, es decir si no hay ningún hilo suspendido esperando a ser ejecutado, simplemente se reinicia el contador *timeslice* del hilo y se termina la ejecución del método de forma que el hilo se sigue ejecutando. En cambio si hubiera algún hilo esperando en la cola *RunnableThread*, se salvaguardan los registros actuales de la maquina virtual en el hilo *CurrentThread* y se pone a NULL para indicar que no hay hilo ejecutándose en ese momento:

```
if (RunnableThreads == NULL) {
    Timeslice = CurrentThread->timeslice;
    return TRUE;
} else {
    storeExecutionEnvironment(CurrentThread);
    threadToAdd = CurrentThread;
    CurrentThread = NIL;
}
```

Una vez configurado la situación para el intercambio de hilos de ejecución se procede a dicha tarea sacando de la lista circular *RunnableThreads* mediante la operación *removeQueueStart* el siguiente hilo a ser ejecutado. Se añade el hilo que estaba ejecutándose anteriormente (y cuya referencia se copio previamente en *threadtoAdd*) a la lista de hilos en espera mediante el método *addThreadQueue* y al final de dicha lista.

Si no se ha obtenido ningún hilo a ejecutar de la lista *RunnableThread* se devuelve un valor lógico falso que indica que no existe ningún hilo para ejecutar. Si existe dicho hilo se cargan los registros de la maquina virtual (*ip,fp,…*) los valores almacenados en el hilo cuando este fue suspendido. Además se reinicia como es lógico

el contador *timeslice* del hilo puesto que había llegado a cero la ultima vez que se había ejecutado:

```
if (CurrentThread == NIL) {
      return FALSE;
}
loadExecutionEnvironment(CurrentThread);
Timeslice = CurrentThread->timeslice;
```

Finalmente se realizan determinadas tareas asociadas a operaciones de mantenimiento y *debug* como es incrementar el contador *ThreadSwtichCounter* que cuenta el número de veces que se produce la operación *thread switching* y marcar en la traza de ejecución esta operación:

```
#if ENABLEPROFILING
    ThreadSwitchCounter++;
#endif

#if INCLUDEDEBUGCODE
if (tracethreading) {
  /* Diagnostics */
  TraceThread(CurrentThread, "Switching to this thread");
}
#endif
```

Por último y antes de que el hilo nuevo empiece a ser ejecutado se comprueba si esta tenía alguna excepción nativa pendiente en cuyo caso se recupera dicha excepción y se eleva mediante el método *raiseExeption*:

```
#if ASYNCHRONOUS_NATIVE_FUNCTIONS
    if (CurrentThread->pendingException != NIL) {
        char *pending = CurrentThread->pendingException;
        CurrentThread->pendingException = NIL;
        raiseException(pending);
    }
#endif
```

13.8. Implementación operaciones creación e inicialización módulo planificación de hilos.

Dentro de este apartado trataremos de estudiar todas las operaciones del módulo de gestión de hilos que están involucradas en la inicialización tanto del propio módulo como de las estructuras de datos principales que manejan.

13.8.1. Creación y finalización de hilos.

Existen dos operaciones públicas que realizan las tareas de creación y finalización de los hilos de programación a nivel de la maquina virtual puesto que como ya hemos comentado con anterioridad la implementación Java del hilo se encuentra en una estructura interna a *THREAD* pero separada de esta.

13.8.2. Creación del hilo.

La creación e inicialización de un hilo a partir de la implementación Java del mismo se realiza a través del método *BuildThread()* que recibe como parámetro precisamente la estructura *JAVATHREAD* a partir de la cual crear la estructura *THREAD* que será devuelta como parámetro.

La primera tarea consiste en crear la estructura THREAD del hilo y la pila de ejecución asociada a él. Dada la criticidad de esta operación primero se crea una estructura temporal (*temporary roots*) para almacenar el hilo reservando la memoria necesaria para tal tarea:

```
DECLARE_TEMPORARY_ROOT(THREAD,
newThreadX,(THREAD)callocObject(SIZEOF_THREAD, GCT_THREAD));
```

Se puede obtener una descripción mas detallada del funcionamiento y uso de estas estructuras temporal así como de las funciones de reserva de memora (*callocObject*) se recomienda la consulta del capítulo *12 Gestión de memoria y recolector de basura*. De esta forma *threadX* es un objeto con la estructura THREAD y de tipo *GCT_THREAD* (tipo interno del recolector de basura). Además se crea una nueva pila de ejecución STACK y se asocia al hilo:

```
STACK newStack = (STACK)callocObject(sizeof(struct stackStruct)
                                                    / CELL,
GCT_EXECSTACK);
  /* newStack->next = NULL;  Already zero from calloc */
  newStack->size = STACKCHUNKSIZE;
  newThreadX->stack = newStack;
```

Antes de asociar a la referencia newThread el hilo temporal creado se añade a la traza de depuración la operación de creación del hilo:

Seguidamente se configuran los parámetros del hilo recién creado, iniciando el valor del contador *timeSlice* al valor por defecto BASERIMESLICE (configurable), se asocia el objeto *javaThread* que representa el hilo Java al hilo interno recién creado y se inicializa el estado del mismo a *THREAD_JUST_BORN* (hilo recién creado):

```
newThread->timeslice = BASETIMESLICE;


javaThread = unhand(javaThreadH);
newThread->javaThread = javaThread;
javaThread->VMthread = newThread;


newThread->state = THREAD_JUST_BORN;
```

Finalmente se añade el hilo creado al principio de la lista *AllThreads* donde se almacenan todos los hilos creados en el sistema devolviendo el puntero al hilo creado (*newThread*):

```
newThread->nextAliveThread = AllThreads;
AllThreads = newThread;
```

```
return(newThread);
```

13.8.3. Finalización del hilo.

La finalización del hilo corre a cargo del método *DismantleThread* que se encarga de liberar todos los recursos asociados al hilo. Todo el método se aplica en un contexto temporal, es decir se asocia el hilo a eliminar a una estructura de tipo raíz temporal mediante la macro *IS_TEMPORARY_ROOTS*.

Lo primero es fijar el estado del hilo en *THREAD_DEAD* (hilo muerto):

```
thisThread->state = THREAD_DEAD;
```

Seguidamente se comprueba la posición del hilo a finalizar dentro de la lista de hilos disponibles en el sistema *AllThreads*. Si el hilo en cuestión es el primero que figura en la lista se elimina de la misma actualizando su referencia al siguiente elemento de la lista. Si por el contrario el hilo a finalizar no es el primero de la lista se busca el hilo en la lista y se elimina de ella nuevamente actualizando la referencia al siguiente elemento de la misma:

```
if (AllThreads == thisThread) {
      AllThreads = AllThreads->nextAliveThread;
} else {
      THREAD prevThread = AllThreads;
      while (prevThread->nextAliveThread != thisThread) {
            prevThread = prevThread->nextAliveThread;
       }
      prevThread->nextAliveThread = thisThread->nextAliveThread;
}
```

Una vez el hilo no figura el la lista de hilos *AllThreads* se marcan como nulos algunos de sus parámetros como la pila de ejecución o las copias de los registros FP y SP:

```
thisThread->nextAliveThread = NULL;
thisThread->stack = NULL;
thisThread->fpStore = NULL;
thisThread->spStore = NULL;
```

Finalmente se comprueba en la lista de temporización de hilos *TimerQueue* si se encuentra dicho hilo, es decir si esta pendiente de ser avisado en un tiempo determinado. En ese caso se elimina la alarma asociada a dicho hilo. Todo lo referente a la temporización de hilos se verá en un apartado posterior:

```
if (inTimerQueue(thisThread)) {
        removePendingAlarm(thisThread);
          }
```

13.8.4. Inicialización del módulo.

Para la inicialización del módulo de gestión de hilos que estamos estudiando se emplea en el inicio de la maquina virtual el método *InitializeThreading* que recibe dos parámetros:

- Un puntero a la clase principal (*mainClass*).
- Los distintos argumentos con que se invocan dicha clase (*arguments*) y que son pasados por línea de comandos al ejecutarla.

Mediante este método se crea el hilo principal a partir del cual se ejecutan y crean el resto de los hilos así como las estructuras necesarias para hacer este módulo compatible con el funcionamiento del recolector de basura.

La primera tarea que se realiza en esta operación es guardar y marcar los parámetros pasados como elementos raíces temporales para que el recolector de basura los trate como tal hasta que el hilo principal se haya inicializado:

```
    DECLARE_TEMPORARY_ROOT(ARRAY, arguments, argumentsArg);
    DECLARE_TEMPORARY_ROOT(JAVATHREAD, javaThread,
(JAVATHREAD)instantiate(JavaLangThread));
```

Seguidamente se marcan como objetos raíces del sistema el puntero al hilo principal *MainThread* (inicializándolo a NULL), el puntero al hilo de ejecución actual (*CurrentThread)*, el puntero a la lista de hilos de espera activa (*RunnableThreads*)y el puntero a la cola de temporizaciones de hilos (*TimerQueue*):

```
makeGlobalRoot((cell **)&MainThread);
MainThread = NULL;
MonitorCache = NULL;
makeGlobalRoot((cell **)&CurrentThread);
makeGlobalRoot((cell **)&RunnableThreads);
makeGlobalRoot((cell **)&TimerQueue);
```

A continuación se inicializa el hilo principal haciendo de la función de creación de hilos y fijando la prioridad por defecto del mismo. Se marca que solo hay un hilo a ejecutar (*AliveThreadCount*), se reinicia el *timeslice* del hilo y pone en estado de espera a ser ejecutado:

```
javaThread->priority = 5;

MainThread = BuildThread(&javaThread);

MainThread->nextThread = NIL;
AliveThreadCount = 1;
Timeslice = BASETIMESLICE;
MainThread->state = THREAD_ACTIVE;
```

Igualmente se inicializa el puntero *CurrentThread* para que apunte al hilo principal que es el primero en ejecutarse y la cola de hilos de espera activa. Este es el momento en el cual se inicializan también los registros de la maquina virtual:

```
CurrentThread = MainThread;
RunnableThreads = NULL;
```

```
TimerQueue = NULL;

setSP((MainThread->stack->cells - 1));
setFP(NULL);
setIP(KILLTHREAD);
```

Como se puede observar se fija la cola de ejecución (*SP*) a partir de la cola almacenada en el propio hilo *MainThread* y no se especifica el marco de ejecución que se ha de emplear puesto que el método del cual arranca el hilo (contenido en la clase *JavaThread*) aún no ha sido inicializado. Si incluyéramos el marco de ejecución de un método no inicializado el recolector de basura podría no funcionar correctamente.

La solución a este problema es crear un marco de ejecución virtual (*pseudo-frame*) invocando al intérprete para que este se encargue de normalizan el método mediante las funciones *pushFrame* y *pushStackAsType*.

```
pushFrame(RunCustomCodeMethod);
pushStackAsType(CustomCodeCallbackFunction,
                  initInitialThreadBehaviorFromThread);
```

Como se puede observar añadimos al marco de ejecución el método *CustomCodeCallbackFunction* de tipo *initInitialThreadBehaviorFromThrebad*. Este último método básicamente realiza las siguientes operaciones:

- Se obtiene la clase principal a partir de la instancia de la misma (función *secondStackAsType*) y se busca el método principal (*main*) a ejecutar (función *getSpecialMethod*) indicando al usuario si se ha producido algún error al buscar el método:

```
thisClass = secondStackAsType(INSTANCE_CLASS);

thisMethod = getSpecialMethod(thisClass, mainNameAndType);
 if (thisMethod == NULL) {
      AlertUser(KVM_MSG_CLASS_DOES_NOT_HAVE_MAIN_FUNCTION);
      stopThread();
} else if ((thisMethod->accessFlags & ACC_PUBLIC) == 0) {
        AlertUser(KVM_MSG_MAIN_FUNCTION_MUST_BE_PUBLIC);
        stopThread();
```

- Una vez encontrado el método se reinicializan los registros del hilo para el nuevo método:

```
DECLARE_TEMPORARY_ROOT(ARRAY, arguments, topStackAsType(ARRAY));
setSP((CurrentThread->stack->cells - 1) + thisMethod->argCount);
setFP(NULL);
setIP(KILLTHREAD);
pushFrame(thisMethod);
((ARRAY *)getLP())[0] = arguments;
```

- Y en caso de que el método sea sincronizado se graba el monitor correspondiente para regular el acceso al mismo:

```
if (thisMethod->accessFlags & ACC_SYNCHRONIZED) {
      getFP()->syncObject = (OBJECT)thisClass;
```

```
        monitorEnter((OBJECT)thisClass);
    } else {
        getFP()->syncObject = NULL;
    }
```

Finalmente y volviendo al método de inicialización principal se añaden a la pila de ejecución la clase principal a ejecutar y los argumentos junto con su tipo y se inicializa la clase:

```
pushStackAsType(INSTANCE_CLASS, mainClass);
pushStackAsType(ARRAY, arguments);
initializeClass(mainClass);
```

Para obtener información detallada de cómo se realizan operaciones sobre clases y objetos tales como el funcionamiento del método initializeClass se recomienda la lectura del capítulo *7 Módulo de estructuras de ejecución internas* y lo referente a los marcos de ejecución de los hilos se encuentra contenido en el capítulo *9 Intérprete de bytecodes*.

13.9. Implementación operaciones activación/desactivación hilos.

En este apartado se tratará de explicar de forma detallada las distintas operaciones del módulo de gestión de hilos relacionadas de forma directa con la parada y arranque de los hilos que componen los programas que se ejecutan en la KVM.

13.9.1. Activación de los hilos.

Por activación del hilo se entiende el mecanismo por el cual el hilo ya creado pasa a un estado pendiente de ser ejecutado. Es decir mediante esta operación se activa el hilo pero este e queda en estado suspendido y se realiza mediante el método *startThread* y que se llama por primera vez al método Java *run* del hilo:

```
void startThread(THREAD thisThread)
```

Primero se marca el hilo en estado suspendido y se incrementa el contador de hilos activos en el sistema:

```
    thisThread->state = THREAD_SUSPENDED;
    AliveThreadCount++;
```

El resto de tareas que se acometen en este método están relacionadas con tareas para depurar de forma correcta el sistema mutihilo de que dispone la KVM.

13.9.2. Detención y eliminación de los hilos.

Mediante el método *stopThread* se detiene la ejecución de hilo que se este ejecutando actualmente y se liberan todos los recursos asociados. De esta forma este método realiza las siguientes operaciones:

- Se suspende la ejecución del hilo llamando al método *suspendThread().*
- Se fuerza la finalización del hilo actual para asegurarse que deja de de ejecutarse:

```
CurrentThread = NIL;
```

- Se fija el estado del hilo en *THREAD_DEAD* para que no se ejecuta más y se decrementa el contador de hilos activos:

```
thisThread->state = THREAD_DEAD;
AliveThreadCount--;
```

- Finalmente se invoca el método DismantleThread para eliminar los recursos asociados al hilo:

```
DismantleThread(thisThread);
```

13.9.3. Suspensión del hilo.

Por suspensión del hilo se entiende suspender la ejecución del hilo actual pero sin detenerlo es decir dejarlo en espera de ser ejecutado de nuevo. Para esta tarea se emplea la función *suspendThread*:

- Primero se comprueba si hay algún hilo ejecutándose. Si no fuera así no habría nada que hacer más:

```
if (CurrentThread == NULL) {
  /* This shouldn't happen, but don't crash */
  return;
 }
```

- Se guardan dentro de hilo una copia de los registros de la KVM en ese momento (para su posterior recuperación cuando se vuelva a ejecutar el hilo). Finalmente mediante el método *signalTimeToReschedule* se fija el *timeslice* del hilo a cero para así forzar una operación *thread switch*:

```
if (!(CurrentThread->state & THREAD_SUSPENDED)) {
  /* Save the VM registers of the currently executing thread */
  storeExecutionEnvironment(CurrentThread);

  /* Signal time to reschedule the interpreter */
  signalTimeToReschedule();
    }
```

- Además se fija el estado del hilo en THREAD_SUSPEND y se borra la referencia a CurrentThread:

```
CurrentThread->state |= THREAD_SUSPENDED;
CurrentThread = NIL;
```

Adicionalmente a este método existe otro denominado *suspendSpecificThread* que recibe como parámetro la referencia (*thread*) del hilo que se desea suspender. Si

dicho hilo resulta ser precisamente *CurrentThread* entonces se realizan las mismas operaciones que en el método *suspendThread* anteriormente comentado. En cambio si el hilo no es el que se esta ejecutando, es decir el referenciado por *CurrentThread* y el estado del mismo es distinto de *THREAD_SUSPEND* o *THREAD_DEAD* se elimina dicho hilo de la cola de hilos activos *RunnableThreads*:

```
if (!(thread->state & (THREAD_SUSPENDED | THREAD_DEAD))) {
      removeFromQueue(&RunnableThreads, thread);
}
```

Este último método solo se emplea en tareas de depuración de los programas Java de ahí que el estado en queda el hilo después de ejecutarse es *THREAD_DBG_SUSPEND*:

```
thread->state |= THREAD_DBG_SUSPENDED;
thread->debugSuspendCount++;
```

13.9.4. Ejecutar hilo.

Mediante la función *resumeThread* que recibe como parámetro un hilo (*thread*) se configura el módulo para que ese hilo pase a ejecutarse cuando haya una operación de *switching* entre hilos:

- Primero se realiza una comprobación para ver si el hilo se encuentra suspendido, si no fuera así se genera un error fatal para indicar al usuario de dicha situación:

```
if (!(thisThread->state & THREAD_SUSPENDED)) {
  fatalError(KVM_MSG_ATTEMPTING_TO_RESUME_NONSUSPENDED_THREAD);
    }
```

- Se actualiza el estado del hilo a *THREAD_ACTIVE* y se comprueba si el hilo que se quiere reanudar se corresponde con el que se esta ejecutando lo cual genera el correspondiente error fatal:

```
thisThread->state = THREAD_ACTIVE;

if (thisThread == CurrentThread) {
      fatalError(KVM_MSG_ATTEMPTING_TO_RESUME_CURRENT_THREAD);
} else {
      addThreadToQueue(&RunnableThreads, thisThread, AT_END);
}
```

- Finalmente mediante el método *addThreadToQueue* se añade el hilo al final de la lista *RunnableThreads* para que cuando se invoque la operación *switchThread* desde el intérprete este hilo sea ejecutado en su momento.

Existe una variante de este método que es *resumeSpecificThread* y que se emplea únicamente para pasar a la lista de hilos a ejecutar un hilo que había sido suspendido desde el depurador.

13.9.5. Funciones auxiliares.

Existen además funciones auxiliares relacionadas con la activación/desactivación de hilos que son:

- *activeThreadCount*: devuelve el número de hilos activos en el sistema.
- *isActivated*: que nos indica si el hilo que le es pasado como parámetro esta activo o no.
- *removeFirstRunnableThread*: invoca internamente a la operación *removeQueueStart(&RunnableThreads)* para de esta manera sacar (que no solo tomar) de la lista de hilos activos el primer hilo. Este método se emplea en la operación de *thread-switching*.

13.10. Implementación operaciones sincronización hilos.

Ya se ha comentado en la introducción de este capítulo como la programación *multihilo* permite ejecutar de forma simultánea varias tareas. Además dichas tareas pueden compartir información mediante determinados mecanismos de sincronización entre ellos.

En Java los hilos comparten información entre ellos mediante objetos que son comunes a estos hilos y que normalmente suelen crearse en el hilo principal (*main*). Para poder controlar el acceso múltiple a estos objetos la KVM dispone de una implementación general e independiente de la maquina sobre la que se ejecuta de lo que se conocen como monitores.

Los monitores son estructuras adicionales asociadas a cada instancia de cualquier objeto o bien la instancia de la clase si el recurso que se comparte dentro del objeto es estático. Estos monitores mantienen una cola en la cual se van almacenando todos los hilos que deseen acceder al objeto y que en el momento de hacerlo este ocupado por otro hilo. Esta cola es del tipo FCFS es decir una vez el objeto queda libre, se permite el acceso al hilo que entró en primer lugar en la cola. Así cualquier hilo que desee acceder al objeto debe comprobar primero el estado en que se encuentra el monitor y si esta ocupado se suspende quedando en espera.

La versión 1.0.3 de la KVM incorpora una serie de modificaciones al monitor que tenía anteriormente la maquina virtual. Esta modificación consiste básicamente en que solo se crean monitores y se alojan en los correspondientes objetos en el caso en el que de verdad sean necesarios. Es decir si solo un hilo hace uso del objeto no se implementa monitor alguno en él. Además se ha añadido un campo mhc asociado a los objetos que almacena información rápida acerca del estado del monitor. Es evidente que estas dos modificaciones obedecen principalmente a una mejor de rendimiento tanto en memoria (por la primera modificación) como en tiempo de acceso al objeto (por la segunda modificación).

La estructura que se emplea para representar el monitor se denomina *monitorStruct* y es accedida a través del identificador MONITOR. Los elementos que integran la estructura son:

- *THREAD owner:* puntero al hilo que esta actualmente en posesión del monitor.
- *THREAD monitor_waitq*: puntero a lista donde se encuentran almacenados los hilos que están suspendidos esperando tomar el monitor.
- *hashCode*: entero largo que representa el *hashcode* del objeto compartido.
- *Depth*: entero que indica el número de veces que el monitor ha sido ocupado recursivamente.
- *OBJECT object*: puntero al objeto compartido, empleado solo para depuración y no disponible en entornos de producción (opción INCLUDEDEBUGCODE no activa).

En la cabecera de los objetos que emplea la KVM internamente figura un campo *mhc* añadido en la versión 1.0.3 que como ya hemos indicado indica el estado del monitor asociado a ese objeto. Los posibles valores que puede tener este campo se encuentran en la lista siguiente:

```
enum MHC_Tag_TypeCode {
    MHC_UNLOCKED  = 0,
    MHC_SIMPLE_LOCK  = 1,
    MHC_EXTENDED_LOCK = 2,
    MHC_MONITOR  = 3
      };
```

Dependiendo del valor que tenga el campo *mhc* los primeros 30 bits de la cabecera del objeto tendrán un significado u otro. Así para los valores *MHC_SIMPLE_LOCK* y *MHC_EXTENDED_LOCK* los primeros 30 bits del objeto apuntan al hilo que bloquea el objeto. Si el valor es *MHC_MONITOR* estos primeros 30 bits referencian al monitor y no al hilo y si el valor es *MHC_UNLOCKED* quiere decir que el monitor no esta ocupado. Para obtener una descripción detallada de la representación interna que la maquina virtual emplea para los objetos se recomienda la lectura del capítulo *7 Módulo de estructuras de gestión interna*.

El flujo de ejecución normal que sigue un hilo que tiene algún elemento en su interior sincronizado es el siguiente:

- Crea el hilo.(CREATED)
- Inicia la ejecución del hilo. (RUNNING).
- Se elimina el hilo cuando llega el comando de terminación del mismo.
- Una vez que se llega al elemento sincronizado se produce el comando de entrada al monitor que lo crea si no existe (ENTERING MONITOR).
- Ejecuta el monitor para que el elemento quede bloqueado en el hilo pudiéndose dar diversos casos:
    - Que termine de ejecutarse el elemento o el hilo por el motivo que sea en cuyo caso se manda la operación para que el monitor quede libre (LEAVING MONITOR).
    - Que se quede esperando que el monitor quede libre (WAITING CONDITION).
    - Que mande una señal a otro hilo indicándole que ya puede ocupar el monitor si hubiera hilos esperando a entrar en el (SIGNALING CONDITION).

Las dos primeras fases han quedado comentadas en los apartados anteriores pues se refieren a tareas relacionados con los hilos en sí, mientras que el resto son operaciones relacionados con la gestión de hilos.

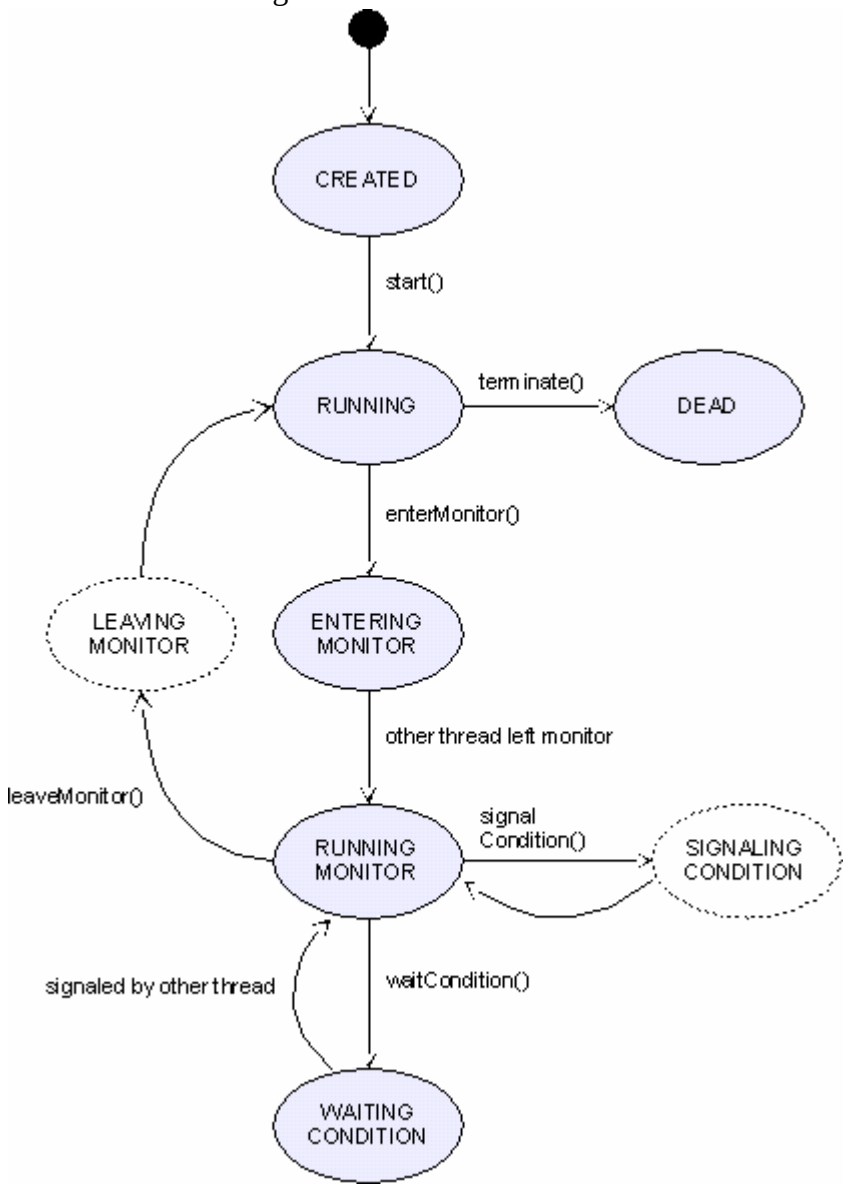

Figura 13.3: Uso de monitores KVM.

Veamos a continuación todas las operaciones relativas al monitor que se emplea en la KVM para sincronizar objetos entre hilos, monitor presente en cada uno de los objetos compartidos entre todos los hilos.

13.10.1. Creación y grabado del monitor.

Para crear un monitor en un objeto compartido se emplea el método *monitorEnter* que recibe como parámetro el puntero al objeto en el cual se quiere grabar dicho monitor (*object*). Este método es invocado cuando el hilo que se esta ejecutando en ese momento trata de acceder a algún objeto Java que sea visible para el resto de hilos del sistema. Este método también se emplea en la operación de grabado de un

monitor que consiste en ocupar dicho monitor y establecer el cerrojo adecuado para que el resto de los hilos que quieran acceder a él queden bloqueados.

Dependiendo del estado en que se encuentre el monitor del objeto se realiza una tarea u otra, es decir tenemos un selector *switch* en base al estado del monitor. Dicho estado se captura haciendo uso de la macro *OBJECT_MHC_TAG*.

Si el estado del monitor es *MHC_UNLOCKED*, nos podemos encontrar con dos situaciones. La primera de ellas sería que el *hashCode* fuese cero, en este caso simplemente se establece una cerradura simple para el objeto y se devuelve el estado *MonitorStatusOwn* para indicar que ha sido creado y ocupado:

```
if (value == MHC_UNLOCKED) {
      SET_OBJECT_SIMPLE_LOCK(object, thisThread);
      return MonitorStatusOwn;
}
```

Para crear este cerrojo simple se emplea la macro *SET_OBJECT_SIMPLE_LOCK*. Esta macro simplemente actualiza el campo dirección del monitor del objeto con la dirección del hilo en cuestión.

```
#define SET_OBJECT_SIMPLE_LOCK(obj, thr) \
            SET_OBJECT_MHC_ADDRESS(obj,      (char      *)(thr)      +
MHC_SIMPLE_LOCK)
```

La segunda situación sería que ya existiese un *hashCode* en el objeto, en este caso se necesita establecer para el objeto un cerrojo rápido (*fast lock*) de profundidad 1 pues no se establece sobre un cerrojo simple anterior y se devuelve *MonitorStatusOwn*:

```
else if (allocateFastLock(thisThread, object, 1, value)) {
         return MonitorStatusOwn;
}
```

Como se puede observar la creación de este cerrojo rápido lo realiza la función auxiliar *allocateFastLock* cuya descripción detallada se puede ver en el apartado dedicado a funciones auxiliares. Si no se da ninguna de las dos opciones anteriores se ejecuta el método auxiliar *upgradeToRealMonitor*.

Si al intentar grabar el monitor este se encuentra en estado *MHC_SIMPLE_LOCK* se comprueba si el cerrojo que tiene el monitor se debe al propio hilo actual, en ese caso hay que actualizar el cerrojo simple a un cerrojo rápido extendido con profundidad (*depth*) 2:

```
      if (OBJECT_MHC_SIMPLE_THREAD(object) == thisThread) {
            if (allocateFastLock(thisThread, object, 2, 0)) {
                        /*Diversas tareas de depuración
                         */
                        return MonitorStatusOwn;
            }
      }
```

Igual que sucedía en el caso anterior si el objeto estuviera bloqueado por otro hilo que no fuera el que se esta ejecutando actualmente o bien no se puede establecer el

cerrojo rápido se ejecuta el método auxiliar *upgradeToRealMonitor* para regularizar esta situación.

Si el estado del monitor es MHC_EXTENDED_LOCK, igual que sucedía con el cerrojo simple se comprueba si es el propio hilo quien tiene el cerrojo extendido aplicado. Si no es así se ejecuta el método upgradeToRealMonitor para obtener el monitor actualizado. En cambio si es el propio hilo quien ya tenia un cerrojo establecido anteriormente simplemente se aumenta la profanidad de dicho cerrojo en una unidad:

```
if (OBJECT_MHC_EXTENDED_THREAD(object) == thisThread) {
      thisThread->extendedLock.depth++;
      return MonitorStatusOwn;

}
```

Si el estado del monitor fuera cualquier otro incluido el MHC_MONITOR se toma del objeto pasado como parámetro el monitor asociado pues dicho monitor ya esta grabado:

```
monitor = OBJECT_MHC_MONITOR(object);
```

Llegado a este punto en la variable monitor tenemos el monitor que debería tener el objeto y *thisThread* contiene la referencia al hilo actual *CurrentThread*. A este punto solo se puede llegar si se ha invocado *upgradeToRealMonitor* (se ha detectado una situación anómala) o bien el tipo del monitor inicialmente era *MHC_MONITOR*. Es el momento de actualizar ciertas variables del monitor en función de quien sea el propietario del monitor.

Si no tiene propietario el monitor, se actualiza el propietario del mismo al hilo actual y se fija la profundidad del cerrojo en 1 y se devuelve *MonitorStatusOwn*:

```
monitor->owner = thisThread;
monitor->depth = 1;
return MonitorStatusOwn;
```

Si el propietario del monitor es el propio hilo simplemente se incrementa la profundidad del cerrojo que ya esta ocupado por el hilo actual y se devuelve *MonitorStatusOwn*:

```
monitor->depth++;
return MonitorStatusOwn;
```

En cambio si el propietario del monitor es otro hilo, se fija el campo *monitor_depth* del hilo actual en 1, se añade el hilo actual a la cola de espera del monitor mediante la función auxiliar *addMonitorWait* y se suspende el hilo devolviendo *MonitorStatusWaiting* y quedando el hilo a la espera de recibir la notificación del monitor para ocuparlo:

```
thisThread->monitor_depth = 1;
addMonitorWait(monitor, thisThread);
suspendThread();
return MonitorStatusWaiting;
```

13.10.2. Finalización ocupación del monitor.

Cuando un hilo esta ocupando un monitor para usar un objeto compartido establece un cerrojo sobre el objeto y el monitor queda grabado como hemos visto. Cuando dicho hilo ha terminado de usar el objeto debe liberar el monitor para que otros hilos puedan usarlo. Dicha operación se ejecuta mediante la invocación el método *monitorExit* que recibe dos parámetros: el objeto compartido (*object*) y un nombre que representa una excepción.

La liberación del monitor implica distintas operaciones dependiendo del estado en el que se encuentre el mismo. Así si el estado (obtenido con la macro *OBJECT_MHC_TAG*) del monitor es *MHC_SIMPLE_LOCK* se realizan las siguientes tareas:

- Primero se comprueba si el hilo que esta ocupando el monitor es distinto de el hilo que se esta ejecutando en ese momento en cuyo caso no se podría liberar el monitor y se devolvería *MonitorStatusError*:

```
if (OBJECT_MHC_SIMPLE_THREAD(object) != thisThread) {
        break;
}
```

- Para liberar el monitor de un cerrojo simple solo hay que inicializar el *hashCode* del objeto y devolver *MonitorStatusRelease*:

```
SET_OBJECT_HASHCODE(object, 0);
return MonitorStatusRelease;
```

Si el estado del monitor es *MHC_EXTENDED_LOCK*, al tener un cerrojo extendido es algo más complejo:

- Primero se comprueba si el hilo que esta ocupando el monitor es distinto de el hilo que se esta ejecutando en ese momento en cuyo caso no se podría liberar el monitor y se devolvería *MonitorStatusError*:

```
if (OBJECT_MHC_EXTENDED_THREAD(object) != thisThread) {
          break;
}
```

- Se decrementa la profundidad extendida del monitor pudiéndonos encontrar con dos situaciones. Si vale cero, es decir la profundidad del cerrojo extendido era 1 simplemente se libera el cerrojo fijando el hashCode del objeto con el hashCode extendido almacenado en el hilo y se libera el cerrojo extendido:

```
int newDepth;
newDepth = --thisThread->extendedLock.depth;
if (newDepth == 0) {
 SET_OBJECT_HASHCODE(object, thisThread->extendedLock.hashCode);
 FREE_EXTENDED_LOCK(thisThread);
 return MonitorStatusRelease;
}
```

- Si la profundidad no es cero, no podemos liberar directamente el cerrojo extendido porque hay por debajo todavía un cerrojo simple, por ello transformamos el cerrojo extendido en cerrojo simple e indicamos que el monitor aun sigue ocupado:

```
if (newDepth == 1 && thisThread->extendedLock.hashCode==0){
      FREE_EXTENDED_LOCK(thisThread);
      SET_OBJECT_SIMPLE_LOCK(object, thisThread);
}
return MonitorStatusOwn;
```

Si el estado del monitor es *MHC_MONITOR*, es decir el *hashCode* del objeto compartido es en realidad un monitor y no un hilo como sucede con los cerrojos anteriores las tareas que se realizan son las siguientes:

- Primero se obtiene el monitor asociado al objeto para actuar sobre él y se comprueba si el propietario del monitor es el hilo que se esta ejecutando:

```
MONITOR monitor = OBJECT_MHC_MONITOR(object);
if (monitor->owner != thisThread) {
         break;
}
```

- Seguidamente se decrementa la profundidad del hilo, si dicha profundidad es cero quiere decir que se puede liberar el monitor. En este caso se invoca la función *removeMonitorWait* (explicada al detalle en el apartado de funciones auxiliares) para notificar a otros hilos que el monitor esta libre. Aquí nos podemos encontrar con dos casos:
  - Que el monitor no lo necesite ningún otro hilo. En este caso se elimina el monitor del objeto y se emplea el slot propietario del monitor extraído para guardar una lista de los monitores aun disponibles:

```
if (   monitor->owner == NULL
        && monitor->monitor_waitq == NULL
        && monitor->condvar_waitq == NULL) {

        SET_OBJECT_HASHCODE(object, monitor->hashCode);

        monitor->owner = (THREAD)MonitorCache;
        MonitorCache = monitor;
}
```

  - Si el monitor lo necesita algún otro hilo simplemente esta disponible para dicho hilo.
- Si la profundidad del monitor es mayor que cero no se puede liberar el monitor y se devuelve *MonitorStatusOwn*.

Evidentemente si el estado del monitor es MHC_UNLOCKED no se hace nada pues ya esta liberado. Si el estado del monitor no se corresponde con ninguno de los comentados anteriormente se señala tal circunstancia rellanando la cadena de excepción que le fue pasada como parámetro:

```
*exceptionName = IllegalMonitorStateException;
 return MonitorStatusError;
```

13.10.3. Espera del hilo para notificación monitor.

Mediante el método *monitorWait* un hilo puede realizar una espera activa de la notificación de un monitor para el cual esta esperando a usar. Para realizar esta espera activa es imprescindible disponer de un objeto monitor real y no de los hilos que los ocupan. Por ello lo primero que se hace es obtener dicho objeto haciendo uso de la función auxiliar *upgradeToRealMonitor*:

```
MONITOR monitor = upgradeToRealMonitor(object);
```

Si el propietario de dicho monitor no es el hilo que se esta ejecutando actualmente se eleva la excepción *IllegalMonitorStateException* y se devuelve el código de error *MonitorStatusError*:

```
if (monitor->owner != CurrentThread) {
  raiseException(IllegalMonitorStateException);
  return MonitorStatusError;
 }
```

Si esta activa la *temporización* de hilos se registra una alarma para de esta manera notificar al hilo en cuestión que ya ha sido liberado el monitor:

```
if (ll_zero_gt(delta)) {
  registerAlarm(CurrentThread, delta, monitorWaitAlarm);
}
```

Mediante el método *addConvarWait* recogido en las funciones auxiliares indicamos al hilo que esta en espera de recibir la notificación y se suspende dicho hilo en espera de que vuelva a tener acceso al objeto si lo necesita más:

```
addCondvarWait(monitor, CurrentThread);
suspendThread();

return MonitorStatusWaiting;
```

La espera del hilo para volver a ocupar el objeto se realiza a través de la alarma de temporización que se ha registrado anteriormente junto con la función *addCondvarWait*. Esta temporización permite establecer un mecanismo de seguridad por si la notificación que debe mandar el monitor cuando ha sido liberado no se ejecuta correctamente. De esta forma se fuerza a volver a intentar capturar el monitor una vez la temporización ha vencido independientemente de que haya llegado la notificación o no.

13.10.4. Notificación del monitor a un hilo suspendido.

Ya hemos visto el método *monitorExit* que permitía que un hilo abandonara la ocupación del monitor. Una vez el monitor queda libre este debe notificar a uno de los hilos que tiene en lista de espera que ya pueden ocuparlo. Esto se realiza mediante el método *monitorNotify()*.

En el caso en el que el monitor tenga el estado *MHC_SIMPLE_LOCK* o *MHC_EXTENDED_LOCK* no es posible tener hilos esperando con este tipo de cerrojos tal y como hemos visto en la función *monitorEnter*:

```
case MHC_SIMPLE_LOCK:
case MHC_EXTENDED_LOCK:
if (address != CurrentThread) {
          raiseException(IllegalMonitorStateException);
          return MonitorStatusError;
      }
return MonitorStatusOwn;
```

Sin embargo si el monitor tiene estado *MHC_MONITOR*, se recupera el monitor asociado al objeto y se comprueba que el propietario de dicho monitor sea el hilo actual *CurrentThread*, si no fuera así se elevaría la excepción *IllegalMonitorStateException*:

```
monitor = OBJECT_MHC_MONITOR(object);
if (monitor->owner != CurrentThread) {
       raiseException(IllegalMonitorStateException);
       return MonitorStatusError;
}
```

Llegado a este punto mediante la función auxiliar *removeConvarWait* se notifica a todos los hilos que estaban esperando (dicha espera se realizaba con *addConvarWait* como hemos visto en el apartado anterior) para que uno de ellos pueda ocupar el monitor:

```
removeCondvarWait(OBJECT_MHC_MONITOR(object), notifyAll);
return MonitorStatusOwn;
```

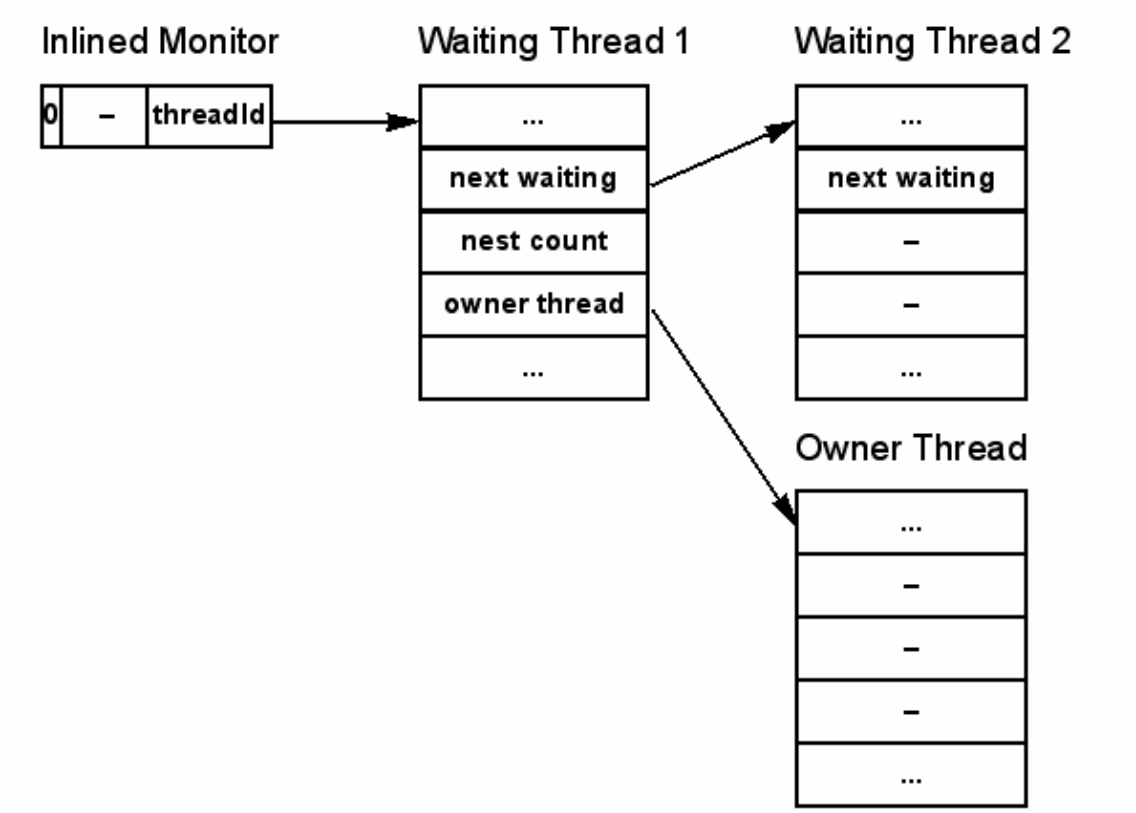

Figura 13.4: Un hilo ocupa un monitor mientras otros dos hilos lo esperan.

13.10.5. Representación de un monitor.

Podemos representar de forma resumida todas las operaciones que hemos comentado anteriormente así como la forma en la que Java representa sus monitores:

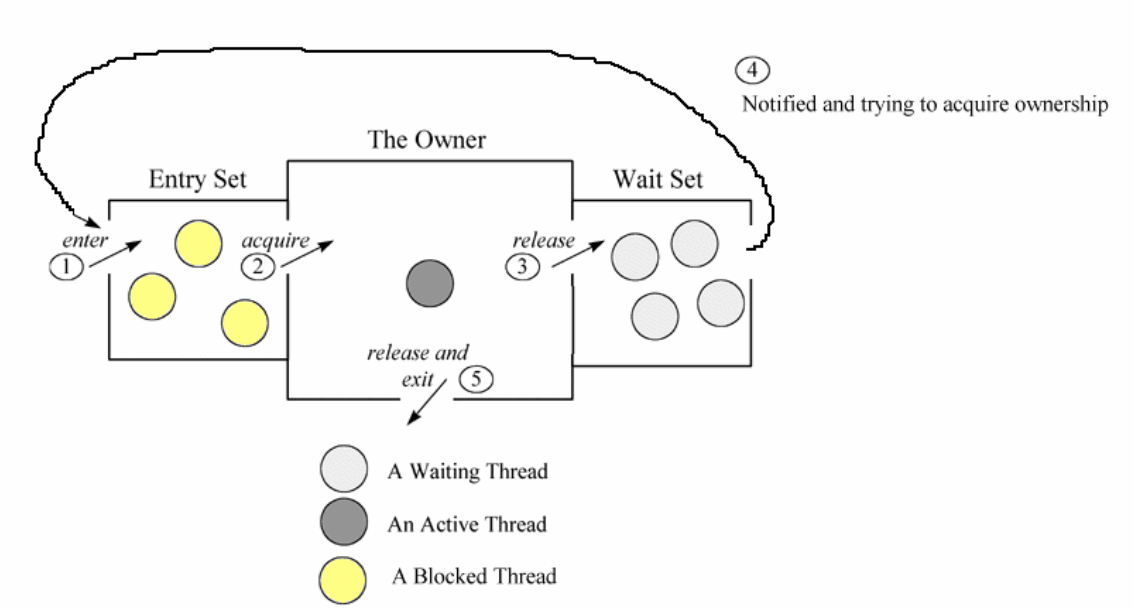

Figura 13.5: Monitor KVM y operaciones asociadas.

13.10.6. Funciones auxiliares.

13.10.6.1. Método *allocateFastLock*.

Este método auxiliar es empleado para fijar un cerrojo rápido a un objeto, lo cual sucede cuando se quiere grabar un monitor desde un hilo y dicho monitor pese a estar en estado UNLOCKED ya posee un valor para el *hashcode* del objeto.

Lo primero que se hace es comprobar si el cerrojo extendido esta libre lo cual se hace mediante la macro *IS_EXTENDED_LOCK_FREE*. Esta macro lo que hace simplemente es comprobar si la profundidad del cerrojo es cero:

```
#define IS_EXTENDED_LOCK_FREE(thread) (thread->extendedLock.depth == 0)
```

Si no se cumpliera la condición anterior no se puede establecer el segundo cerrojo al monitor por lo que se devolvería al punto desde el cual se invoco la función un valor lógico false. Si se cumple la condición, se fija la profundidad del cerrojo y el *hashCode* del cerrojo extendido en el campo *extendedLock* del hilo a partir de los valores que han sido pasados como parámetros a la función:

```
thread->extendedLock.depth = depth;
thread->extendedLock.hashCode = hashCode;
```

Finalmente para establecer el cerrojo de verdad se invoca la macro *SET_OBJECT_EXTENED_LOCK* y se devuelve un valor lógico cierto:

```
SET_OBJECT_EXTENDED_LOCK(object, thread);
return TRUE;
```

Esta última macro funciona de forma idéntica a la que establece el cerrojo simple fijando la referencia al hilo que ocupa el monitor en el espacio del campo *hashCode* adecuado del monitor:

```
#define SET_OBJECT_EXTENDED_LOCK(obj, thr) \
        SET_OBJECT_MHC_ADDRESS(obj,     (char     *)(thr)     +
MHC_EXTENDED_LOCK)
```

Una descripción detallada de algunas de las macros más usuales se puede encontrar en el ultimo apartado. Como se ha podido comprobar este cerrojo extendido se basa en guardar la información del mismo en el hilo que lo ocupa, así el campo *extendedLock* del hilo guarda la información del segundo cerrojo que hay que aplicar cuando sea necesario.

13.10.6.2. Método *upgradeToRealMonitor*.

Este método es invocado en varios puntos de ejecución de las operaciones de sincronización de hilos para actualizar el monitor cuando este se encuentra en un estado no correcto. Un estado no correcto puede darse por ejemplo si el monitor tiene el estado *MHC_UNLOCKED* y sin embargo hay un valor para el hashCode del objeto compartido que hace que no se pueda establecer ni el cerrojo simple ni el doble.

La ejecución de este método comienza ejecutando obteniendo el tipo de monitor asociado al objeto al objeto que le es pasado como parámetro. Si resulta ser del tipo MHC_MONITOR se devuelve dicho monitor al punto desde el que fue llamado puesto que no hay que realizar ninguna tarea:

```
enum MHC_Tag_TypeCode tag = OBJECT_MHC_TAG(object);
      MONITOR monitor;

      if (tag == MHC_MONITOR) {
            return OBJECT_MHC_MONITOR(object);
      }
```

Si el objeto no lleva en su *hashCode* un monitor, se realiza una segunda comprobación y es ver si hay algún monitor almacenado en la cache (*MonitorCache*) esperando a ser incorporado a algún objeto. Si hay un monitor en cache se copia la referencia a ese monitor en el monitor que se quiere crear o modificar (monitor), se actualiza el *monitorCache* y se inicializa el hilo propietario, el *hashCode* y el campo *depth*:

```
monitor = MonitorCache;
MonitorCache = (MONITOR)monitor->owner;
monitor->owner = NULL;
monitor->hashCode = 0;
monitor->depth = 0;
```

Si por el contrario no hay ningún monitor en *cache* se crea el monitor dentro de una región crítica (para protegerlo de la acción del recolector de basura):

```
START_TEMPORARY_ROOTS
    DECLARE_TEMPORARY_ROOT(OBJECT, tObject, object);
    monitor = (MONITOR)callocObject(SIZEOF_MONITOR,GCT_MONITOR);
    object = tObject;
END_TEMPORARY_ROOTS
```

Una vez obtenido el monitor ya sea de la cache o creado desde cero hay que actualizar los valores del mismo. Para ello se emplea un selector tipo *switch* en base al tipo de monitor que se extrajo al principio del método.

Si el tipo es MHC_UNLOCKED dado que el objeto no esta cerrado se actualiza el *hashCode* del monitor:

```
monitor->hashCode = object->mhc.hashCode;
```

Si el tipo es MHC_SIMPLE_LOCK se actualiza el propietario del monitor extrayendo el mismo de objeto a través de la macro correspondiente:

```
monitor->owner = OBJECT_MHC_SIMPLE_THREAD(object);
monitor->depth = 1;
```

Si el tipo es *MHC_EXTENDED_LOCK* a partir del objeto se extrae el hilo que aplica el cerrojo extendido. Se fija como propietario del monitor a dicho hilo y se actualiza la profundidad (*depth*) y el *hashCode* del monitor a la profundidad y *hashCode* extendidos almacenados en el hilo. Además se libera del hilo el cerrojo rápido pues ya ha sido grabado en el monitor:

```
THREAD thread = OBJECT_MHC_EXTENDED_THREAD(object);
monitor->owner = thread;
monitor->depth = thread->extendedLock.depth;
monitor->hashCode = thread->extendedLock.hashCode;
FREE_EXTENDED_LOCK(thread);
```

Una vez creado el monitor se actualiza el monitor del objeto con este nuevo monitor creado y se devuelve dicho monitor al punto de invocación del método.

```
SET_OBJECT_MONITOR(object, monitor);
return monitor;
```

13.10.6.3. Funciones de espera de un monitor.

Cuando un hilo quiere acceder a un objeto y el monitor de dicho objeto esta bloqueado por otro hilo, se produce la llamada a la función *addMonitorWait* que se encarga de añadir dicho hilo a la lista de espera del monitor. De esta forma el hilo se queda esperando a que el monitor quede libre.

La implementación del método addMonitorWait sería de la siguiente forma:

- Mediante el método addThreadToQueue se añade el hilo que le es pasado como parámetro (thread) a la lista de hilos en espera que posee el monitor que también le es pasado como parámetro (monitor):

```
addThreadToQueue(&monitor->monitor_waitq, thread, AT_END);
```

- Se actualiza el estado del hilo a THREAD_MONITOR_WAIT y se actualiza la copia del monitor del hilo con el monitor en cuestión del objeto al cual se esta accediendo:

```
thread->monitor = monitor;
thread->state |= THREAD_MONITOR_WAIT;
```

- Además si el monitor en cuestión no tiene propietario (es decir no esta ocupado) se ejecuta por seguridad el método *removeMonitorWait*(monitor) que fuerza a que el monitor notifique al siguiente hilo que puede acceder al objeto en cuestión:

```
if (monitor->owner == NULL) {
  removeMonitorWait(monitor);
}
```

Como acabamos de comentar mediante el método *removeMonitorWait* el monitor notifica a alguno de los hilos que estaban esperando en su cola que ya puede ocupar el monitor y bloquear el objeto mediante el cerrojo apropiado. Este método opera de la siguiente forma:

- Obtiene de la lista de hilos en espera mediante el método apropiado el primer hilo que debe ser notificado:

```
THREAD waiter = removeQueueStart(&monitor->monitor_waitq);
```

- Si no hay hilo esperando se fija el propietario del monitor a NULL y su profundidad a cero:

```
monitor->owner = NULL;
monitor->depth = 0;
```

- Si existe algún hilo esperando en la cola, se modifica el propietario del monitor para que sea el nuevo hilo que lo ocupa y se toma la profundidad del monitor del parámetro del hilo:

```
monitor->owner = waiter;
monitor->depth = waiter->monitor_depth;
```

- Además se inicializa el monitor del hilo y su profundidad por seguridad para que pueda ser modificado convenientemente y se activa la ejecución del hilo invocando a la operación *resumeThread()*:

```
waiter->monitor = NULL;
waiter->monitor_depth = 0;
resumeThread(waiter);
```

13.10.6.4. Funciones de notificación de un monitor.

Existen dos funciones empleadas en los métodos principales *monitorWait* y *monitorNotify* y que se usan para indicar que un hilo esta esperando a ser notificado (*addConvarWait*) y para notificar específicamente a un hilo que estabas esperando a un monitor.

La función *addConvarWait* realiza las siguientes tareas:

- Primero comprueba que el propietario del monitor pasado como parámetro coincide con el hilo también pasado como parámetro:

```
if (monitor->owner != thread) {
  fatalVMError(KVM_MSG_BAD_CALL_TO_ADDCONDVARWAIT);
    }
```

- Seguidamente se añade el hilo a la cola *monitor_waitq* del monitor donde se almacenan los hilos que esperan notificación, se salvaguarda el monitor en el hilo y se actualiza el estado a *THREAD_CONVAR_WAIT*:

```
addThreadToQueue(&monitor->condvar_waitq, thread, AT_END);
thread->monitor = monitor;
thread->state |= THREAD_CONVAR_WAIT;
```

- Se salvaguarda también la profundidad del monitor para poder ser recargada posteriormente cuando el hilo recupere el control del monitor nuevamente:

```
thread->monitor_depth = monitor->depth;
```

- Finalmente se realiza una llamada a *removeMonitorWait* para así notificar a algún otro hilo que el monitor esta libre y puede ser ocupado:

```
removeMonitorWait(monitor);
```

Por otro lado la función *removeConvarWait* se completa en los siguientes pasos:

- Se obtiene el primer hilo de la lista condvar_waitq del monitor que le es pasado como parámetro y se implementa un bucle para recorrer la lista hasta que todos los hilos han sido notificados:

```
THREAD waiter = removeQueueStart(&monitor->condvar_waitq);
if (waiter == NULL) {
      break;
 }
```

- Para cada hilo de la lista en caso de que existiera un alarma temporal registrada para dicho hilo se elimina la alarma:

```
removePendingAlarm(waiter);
```

- Se invoca la función *addMonitorWait* para así esperar a poder tomar en propiedad el monitor ya que se ha recibido la notificación de que ha quedado libre:

```
addMonitorWait(monitor, waiter);
```

13.10.6.5. Macros de configuración empleadas.

A lo largo del apartado 10.9 hemos visto muchas operaciones relacionadas con el sincronismo de los hilos de ejecución y en su paso se han empleado algunas macros para simplificar y estandarizar los procesos.

Algunas de estas macros son:

```
#define SET_OBJECT_MHC_LONG(obj, value) (obj)->mhc.hashCode = (value)
#define SET_OBJECT_MHC_ADDRESS(obj, value) (obj)->mhc.address =
(value)
```

Como se puede comprobar estas macros actualizan los campos *hashCode* y long del elemento *mhc* del objeto que le es pasado como parámetro. Esta macro se ha usado por ejemplo para fijar los cerrojos de un monitor tanto el extendido como el simple:

```
#define SET_OBJECT_SIMPLE_LOCK(obj, thr) \
        SET_OBJECT_MHC_ADDRESS(obj, (char *)(thr) + MHC_SIMPLE_LOCK)

#define SET_OBJECT_EXTENDED_LOCK(obj, thr) \
        SET_OBJECT_MHC_ADDRESS(obj, (char *)(thr) + MHC_EXTENDED_LOCK)
```

Se puede ver como fijar un cerrojo u otro se basa un usar la referencia al hilo que aplicara el cerrojo afectado por el tipo de cerrojo para actualizar el *hashCode* del objeto.

Otra de las macros muy usadas son aquellas relacionadas con el tipo de monitor del objeto. Para saber dicho tipo se emplea la siguiente macro:

```
#define OBJECT_MHC_TAG(obj) ((enum MHC_Tag_TypeCode)((obj)-
>mhc.hashCode & 0x3))
```

Que como vemos obtiene el *hashCode* del objeto y lo cruza con la lista de tipos de monitores disponibles para devolver el correspondiente (*MHC_MONITOR*, *MHC_UNLOCKED*,…) Otra de las macros también muy empleada es aquella que permite obtener el monitor asociado a un objeto:

```
#define OBJECT_MHC_MONITOR(obj)  \
       ((MONITOR)(((char *)(obj)->mhc.address) - MHC_MONITOR))
```

Y aquellas macros que a partir de un objeto extraen del monitor del mismo los hilos que han aplicado un cerrojo sobre él ya sea simple o extendido:

```
#define OBJECT_MHC_SIMPLE_THREAD(obj) \
      ((THREAD)(((char *)(obj)->mhc.address) - MHC_SIMPLE_LOCK))
#define OBJECT_MHC_EXTENDED_THREAD(obj) \
    ((THREAD)(((char *)(obj)->mhc.address) - MHC_EXTENDED_LOCK))
```

Otra macro que se ha empleado a lo largo de algunas de las operaciones de hilos es la encargada de liberar el cerrojo rápido o extendido almacenado en el hilo:

```
      #define     FREE_EXTENDED_LOCK(thread)                    (thread-
>extendedLock.depth = 0)
```

Que como se puede ver solo fija la profundidad del cerrojo en cero pues al ser un cerrojo extendido cuando esta en uso su valor mínimo es 1.

13.11. Soporte para operaciones de I/O concurrentes a la planificación de hilos.

Como ya se ha comentado en un apartado anterior a partir de la versión 1.0 de la KVM se incluyo en la misma un soporte para operaciones de entrada/salida relacionado con la gestión de hilos de ejecución. Dicho soporte implica el que se permitan realizar en paralelo operaciones de entrada/salida y ejecuciones de hilos.

Es decir en versiones anteriores de la KVM y para evitar degradaciones de recursos asociados (tales como la memoria del sistema hardware) mientras se ejecutaba cualquier operación I/O se suspendían todos los hilos que estuvieran ejecutándose hasta que la operación finalizara. Pues bien, la versión 1.0 de la KVM para aumentar el rendimiento de la misma permite realizar operaciones I/O mientras se ejecuta algún hilo.

Evidentemente las operaciones asíncronas de I/O solo son válidas si el sistema sobre el cual se ejecuta la maquina virtual provee lo mecanismos necesarios en forma de interrupciones, funciones *callback*, o hilos nativos. Sin embargo, hay que tener en cuenta que el uso de estas operaciones implica un crecimiento sustancial de complejidad.

Cuando el autor de una función nativa Java desea permitir que el interprete sea ejecutado mientras operaciones I/O son realizadas, este debe hacer una llamada a *suspendThread*, fijar el mecanismo apropiado para esperar la respuesta de la operación y devolver el control al sistema. Es muy importante que la llamada al método *suspendThread* se haga siempre antes de que la función *callback* que se use sea ejecutada. Cuando la función *callback* es ejecutada, normalmente dicha función querrá localizar algunos resultados de la función en la pila de ejecución del hilo que se había suspendido, para ello debe llamar a *resumeThread*.

Cuando una función nativa es invocada en la KVM, es ejecutada normalmente dentro del contexto del hilo que realizo la llamada. Sin embargo este no es el caso de las funciones *callback* (o cualquier otro mecanismo que se use para informar de los resultados de una operación I/O) cuando son invocadas. Consecuentemente las funciones y macros normales de acceso a la pila de ejecución tales como *popStack* no pueden ser usadas dentro de la función *callback*. En su lugar se emplean métodos idénticos a estos pero que se identifican pues se le añade al nombre del método *FromThead* al final del mismo, por ejemplo *popStackFromThread*.

Usando estas nuevas rutinas la pila de ejecución de los hilos puede ser accedida y/o modificada con los parámetros apropiados incluso antes de invocar *activateThread*.

Sin embargo hay un problema adicional con las operaciones asíncronas de I/O y es que puede que se produzcan condiciones de carrera. Para prevenir que esta condición pueda corromper la lista *RunnableThreads*, que constituye el enlace entre el mundo asíncrono exterior y las funciones internas de planificación, todos los accesos a esta lista han de estar controlados por una región crítica. Para ello se emplean dos macros *START_CRITICAL_SECTION* y *STOP_CRITICAL_SECTION* que internamente invocan dos funciones propias del sistema *enterSystemCriticalSection()* y *exitSystemCriticalSection()*.

En este apartado trataremos de estudiar en detalle las operaciones que involucran el uso de esta nueva facilidad de la KVM, operaciones que por supuesto son opcionales.

13.11.1. Invocaciones nativas de I/0.

Para ejecutar operaciones asíncronas desde un hilo se pueden emplear dos métodos. El primero de ellos es *asyncFunctionProlog* que permite que se ejecuten funciones nativas asíncronas preparando el sistema para ello de la siguiente forma:

- Se comprueba si se esta ejecutando el recolector de basura, en cuyo caso se aborta la invocación del método pues el hilo sobre el cual se va operar es el hilo que actualmente se esta ejecutando. Si se realizara la invocación simultáneamente con el recolector de basura se podrían provocar acceso múltiples a memoria:

```
if (collectorIsRunning) {

fatalError(KVM_MSG_COLLECTOR_IS_RUNNING_ON_ENTRY_TO_ASYNCFUNCTIO
NPROLOG);
}
```

- Se suspende el hilo que se estaba ejecutando, mediante el método *suspendThread*.

- Seguidamente se marca la entrada en una sección crítica de ejecución pues se simultanean operaciones de I/O con ejecuciones de hilos y se actualiza el contador de funciones nativas asíncronas y se invocaría la función nativa especifica:

```
START_CRITICAL_SECTION
  AsyncThreadCount++;
  INVOCACION_FUNCION_NATIVA
END_CRITICAL_SECTION
```

La otra función que permite la ejecución de operaciones asíncronas es *asyncFunctionEpilog* y que recibe una referencia a un hilo. Mediante esta función se realiza una espera activa hasta que todas las operaciones de I/O han finalizado realizando las siguientes fases:

- Inicia la ejecución del hilo si no estaba activo invocando a la función *resumeThread*.

- Se inicia una sección crítica y dentro de esta se ejecuta la macro *NATIVE_FUNCTION_COMPLETED* que sería la que de verdad implementaría la espera activa. También se decrementa el contador de funciones nativas asíncronas e ejecución igual que en la operación anterior se incremento.

13.11.2. Interacción con el recolector de basura.

Desde las funciones nativas del sistema se puede invocar a dos funciones dentro de este módulo de gestión de hilos para habilitar o deshabilitar el funcionamiento del recolector de basura mientras se ejecuta la función nativa. Estas dos funciones son:

- *decrementAsyncCount*: mediante esta función y decrementando el contador *AsyncThreadCount* dentro de una región crítica se habilita el recolector de basura.
- *incrementAsyncCount*: mediante esta función y decrementando el contador *AsyncThreadCount* dentro de una región crítica se habilita el recolector de basura.

Por otro lado en el algoritmo que el emplea el recolector de basura de la KVM y que puede consultarse en detalle en el capítulo 8 antes de ejecutar el recolector propiamente dicho uno de los pasos que se hace es invocar la función *RundownAsynchronousFunctions*. Igualmente después de aplicado el algoritmo de recolección de basura e ejecuta la función *RestartAsynchronusFunctions*.

Ambas funciones forman parte del módulo de gestión de hilos y como sus nombre indican se encargan de finalizar todas las operaciones nativas de I/O y arrancarlas de nuevo justo después de aplicado el algoritmo del recolector.

La función *RundownAsynchronousFunctions* ejecuta el siguiente flujo en bucle infinito:

- Se comprueba que el recolector de basura no esta funcionando:

```
if (collectorIsRunning) {
fatalError(KVM_MSG_COLLECTOR_RUNNING_IN_RUNDOWNASYNCHRONOUSFUNCT
IONS);
        }
```

- Dentro de una región crítica si el contador de funciones nativas en ejecución (modificado por las funciones anteriormente comentadas en este mismo apartado) es distinto de cero se realiza una espera:

```
START_CRITICAL_SECTION
      if (AsyncThreadCount == 0) {
          collectorIsRunning = TRUE;
      }
 END_CRITICAL_SECTION

  if (collectorIsRunning) {
      break;
```

```
  }


 Yield_md();
```

Dicha espera se implementa en *Yield_md()* y es dependiente de la maquina finalizando cuando se marca *collectorIsRunning* y es cuando han finalizado todas las funciones nativas.

La función *restartAsynchronousFunctions* se implementa con el siguiente flujo:

- Se comprueba que no se ejecuta el recolectar de basura en ese momento.

```
if (!collectorIsRunning) {

      fatalError(KVM_MSG_COLLECTOR_NOT_RUNNING_ON_ENTRY_TO_RESTAR
      TASYNCHRONOUSFUNCTIONS);
}
```

- Se actualiza la variable collectorIsRunning.

```
collectorIsRunning = FALSE;
```

13.12. Funciones auxiliares de manipulación de colas.

Se ha podido comprobar a lo largo del capítulo como el módulo de gestión de hilos emplea para determinados propósitos tales como almacenar hilos en espera de un monitor o de una notificación del mismo colas. Igualmente a lo largo del capítulo se han hecho referencia a una serie de métodos que permitían interactuar con estas colas. Estos métodos son:

- *addThreadQueue*: permite añadir un hilo pasado como parámetro al principio o al final de la lista también pasada como parámetro.
- *removeQueueStart*: permite tomar el primer hilo de la cola pasada como parámetro.
- *removeFromQueue*: permite remover un hilo especifico de la cola devolviendo un valor lógico cierto o falso dependiendo de si dicho hilo estaba presente en la cola o no.

Comentar en detalle estos métodos no merece la pena pues simplemente constituyen métodos básicos de acceso a las listas circulares que simulan las colas antes mencionadas. Lo que si es muy importante mencionar es que internamente esta operaciones se ejecutan dentro de una región crítica que recordemos se invocan mediante las macros *START_CRITICAL_SECTION* y *END_CRITICAL_SECTION*. Si bien se podría eliminar la dependencia de estas operaciones con una región crítica no es aconsejable hacerlo.

Estas colas constituyen el mecanismo básico de sincronismo entre hilos en la KVM junto con los cerrojos de los monitores y por ello se recomienda que los accesos a estas listas este exhaustivamente controlado por el sistema de ahí el que se encuadren dentro de una región crítica.

13.13. Parámetros configurables del planificador de hilos.

Una vez que hemos visto de forma exhaustiva como funciona el módulo de gestión *multihilo* de la KVM vamos a tratar de exponer de forma clara y concisa que parámetros de configuración podemos modificar para adaptar dicho módulo a nuestras necesidades. Sin embargo nos encontramos en que para este módulo no tenemos acceso directo a parámetros de configuración claros, al menos parámetros accesibles desde el fichero main.h.

Si existen algunas opciones de customización relacionadas por ejemplo con el uso de funciones nativas asíncronas y que se han estudiado en este capítulo. Otro de los parámetros que se podría modificar es el tiempo *timeSlice* base recogido en *BASE_TIMESLICE* que indica la porción de tiempo que un hilo esta ejecutándose de manera continuada y que recordemos esta fuertemente relacionado con el funcionamiento del intérprete. Si el sistema final va a estar orientado a la ejecución de muchos hilos conviene disminuir este *timeslice*, si bien este no suele ser un caso habitual en las aplicaciones desarrolladas la KVM.

Existe un parámetro de configuración que afecta al rendimiento de todos los módulos y por consiguiente a este que nos ocupa ahora y es el depurador. La opción *ENABLE_JAVA_DEBUGGER* e *INCLUDE_DEBUG_CODE* introduce mucho código con fines de monitorización, es por ello que esta opción suele estar deshabilitada en entornos de producción y solo se emplea para pruebas de la maquina virtual cuando se ha introducido alguna modificación o se ha detectado un *bug* en la versión que estamos estudiando.

13.14. Conclusiones.

Una vez que se han visto en anteriores capítulos como la maquina virtual de la plataforma J2ME ejecuta una aplicación Java de un hilo, en este capítulo se ha profundizado en la implementación que la KVM hace del sistema *multihilo*.

La estructura que la KVM emplea para representan los hilos se denomina *Thread* que en realidad es una lista enlazada de estructuras del tipo *threadQueue*. Así un hilo estaría formado por un elemento de tipo *Thread* con un elemento en su interior de tipo *threadQueue*. Estos elementos que representan hilos se componen principalmente de los siguientes parámetros:

- *JavaThread*: estructura que almacena la instancia de la clase Java que representa el hilo.
- *Stack*: puntero a la pila de ejecución del hilo.
- Registros de salvaguarda de los valores que estos tienen en la maquina virtual.
- *Monitor* y *extended Lock*: estructuras que se emplean para sincronización entre hilos.
- Estado del hilo.

El procedimiento por el cual la maquina virtual crea nuevos hilos de ejecución conlleva una serie de pasos entre los cuales cabe destacar:

- Reservar memoria para la pila de ejecución del método.
- Actualizar el tiempo base de ejecución del hilo y sus estados en *recién creado*.
- Se añade el hilo a la lista de hilos disponibles en el sistema *AllThreads*.

Cuando un hilo no va a ser usado, es decir para finalizar de forma correcta un hilo Java, se procede a la eliminación de dicho hilo de la lista de hilos disponibles. De esta forma, cuando el recolector de basura sea invocado, al no encontrar referencias a este hilo lo eliminará.

Una de las operaciones más importantes referentes a los hilos es cuando un hilo debe pasar a ejecutarse en detrimento de otro que estuviera ejecutándose. En la KVM esto se logra mediante un mecanismo básico y es que un hilo deja de ejecutarse por dos motivos:

- Que se supere el tiempo de ejecución base del hilo. En este caso el hilo pasa a formar parte de una lista de hilos activos esperando ser ejecutados y el primer hilo de esta lista es el que es ejecutado por el intérprete.
- Que se invoque una operación de suspensión del hilo o finalización del mismo (forzada o no) obteniendo el mismo resultado.

Se han examinado a lo largo del capítulo las múltiples operaciones que la maquina virtual permite que se realicen sobre los hilos: activación, detención de ejecución, eliminación, ejecución de un hilo, reanudación del mismo. Estas operaciones actúan revisando una serie de listas de hilos que mantienen los estados de los mismos. La activación del hilo implica modificar el estado del mismo y ejecutar el método especial *run*. Para detener un hilo previamente se suspende la ejecución del mismo y se finaliza como se ha comentado en el apartado anterior. La suspensión del hilo se realiza modificando el estado del mismo, invocando la operación de intercambio de hilos y se elimina de la lista de hilos dispuestos a ser ejecutados *RunnableThreads*. Así cualquier hilo que se cree cuando ya hay otro hilo ejecutándose pasa a ocupar el último puesto de la lista de hilos *RunnableThreads*.

Finalmente se ha analizado como los distintos hilos que se ejecutan en la maquina virtual pueden compartir objetos entre ellos. A nivel de lenguaje Java los hilos pueden compartir elementos mediante el uso de la palabra reservada synchronized.

En la KVM la sincronización entre objetos se logra mediante un mecanismo de monitorización de los objetos compartidos. De esta forma cuando un objeto es accedido desde un hilo, se graba en el monitor de dicho objeto y en el hilo esta situación. Así cuando un segundo hilo desea acceder al objeto primero comprueba el estado del monitor y si este esta ocupado se pone a la espera en una lista. Cuando el hilo primero termina de ejecutar el método envía una notificación al primer hilo de la lista de espera para que pueda grabar el monitor.

En realidad la KVM no emplea un monitor básico como el que se ha comentado en el apartado anterior sino que emplea un cerrojo triple. El primer cerrojo lo aplica el primer hilo que accede al objeto (grabación del monitor), el segundo cerrojo se produce cuando un segundo hilo desea acceder al objeto (*simple lock*) y el tercer cerrojo (*extended lock*) se produce cuando desde el primer hilo se quiere volver a acceder al

objeto. El tercer cerrojo mantiene una dimensión (*depth*) que indica cuantas veces ha intentado acceder el hilo al objeto.

A la hora de liberar un objeto siempre se empieza desde el cerrojo extendido para cada una de las dimensiones que tenga, teniendo en cuenta que este caso no se envía notificación ninguna a otro hilo. Después se prosigue con el cerrojo simple para finalmente liberar el monitor del objeto compartido.