Máquina virtual de Java multiaplicación

Autores:
Jose Carlos Alján Sanz
Ricardo González Moreno
Rebeca Shakar Serrano

Profesor director:
Katzalin Olcoz Herrero

Proyecto de Sistemas Informáticos
Facultad de informática

Universidad Complutense de Madrid

Curso 2005-2006

Índice

1. Resumen de la memoria

1.1. Resumen en castellano

El primer objetivo de esta memoria es que el lector llegue a tener una visión general de la arquitectura y el funcionamiento de la máquina virtual de Java llamada JikesRVM.

Para el objetivo del proyecto, el sistema de hilos muy importantes. Son el mecanismo que utilizaremos para conseguir la ejecución de más de un programa sobre la máquina virtual. Por eso trataremos su estructura y su implementación. Después explicaremos las modificaciones hechas en esta parte de la máquina virtual para conseguir el objetivo del proyecto.

El gestor de memoria es el otro sistema a modificar. JikesRVM da la opción de utilizar un gestor dentro de todos los que tiene implementado. En nuestro caso usaremos el denominado CopyMS. Estudiaremos la estructura global de los gestores y la implementación de estos. Posteriormente explicaremos los cambios realizados sobre el gestor de memoria en este proyecto.

Por último se hará un estudio sobre el rendimiento y eficiencia conseguidos con las mejoras añadidas a JikesRVM sobre el original. Para ello se usarán benchmarks especiales escritos en lenguaje Java.

1.2. English sumary

Fristly, in this report, we will try to give the reader a general vision of the Java virtual machine's architecture and how it works.

The thread system is very important for the goal of the proyect. They are the mechanism that we will use to achieve the execution of more than one porgram on the virtual machine. This is the reason of we will study and deal with its structure and its implementation. After that we will explain what we have modified in this part of the virtual machine to achieve the aim of the project.

The memory managment is the other system to modify. JikesRVM gives the opportunity to use one of the memory management among all it has implemented. In our case we have chosen the named CopyMS. We will study the global structure of the memory management and its implementation. After that we will explain the changes made in the memory managment.

At last a study about yield and efficiency will be done, comparing the improved machine with the original one. To achieve this special benchmarks written in Java language will be used.

2. Visión global de JikesRVM

JikesRVM es una máquina virtual orientada a servidores, escrita a su vez en el lenguaje de programación Java. Para poder cumplir los requerimientos que dichos servidores necesitan, Jikes ha sido implementado para ser lo más autosuficiente posible.

Una característica importante es que tiene dos compiladores (el "baseline compiler" y el "optimizing compiler") y que no posee interprete.

Con respecto a la memoria, un servidor no tiene tantas restricciones de memoria como pueden llegar a tener otras plataformas. Pero como inconveniente tiene que cumplir ciertos requerimientos, que son propios de los servidores. Estas exigencias son principalmente las siguientes:

Explotación de procesadores de alto rendimiento: los compiladores JIT (just-in-time) no realizan las optimizaciones necesarias para aprovechar al máximo rendimiento de las nuevas características del hardware (por ejemplo el paralelismo del multiprocesador). Estas son absolutamente necesarias para obtener un rendimiento comparable con el de otros lenguajes compilados estáticamente.

escalabilidad SMP: la configuración SMP (shared-memory multiprocessor) es muy utilizada en servidores. En algunas máquinas virtuales, los programas multihilo Java conllevan una pobre escalabilidad, aumento considerable en el número de hilos.

Limites de hilos: muchos servidores necesitan crear nuevos hilos para cada petición. No obstante debido a restricciones del sistema operativo, algunas máquinas virtuales no pueden designar un gran número de hilos, por lo tanto solo pueden tratar con un número limitado de peticiones a la vez. Estas restricciones limitan a las aplicaciones soportar cientos de usuarios simultáneos.

Disponibilidad continua: las aplicaciones de los servidores, deben satisfacer las peticiones que se le realizan. Teniendo en cuenta que dichas aplicaciones funcionan continuamente durante largos periodos (algunos meses). Esta disponibilidad continua, no aparece dentro de las prioridades de las actuales máquinas virtuales.

Respuesta rápida: muchas aplicaciones de los servidores tienen requerimientos muy estrictos en lo que a tiempo de respuesta se refiere. Sin embargo, muchos de los recolectores de basura de las actuales máquinas virtuales de Java, son los responsables del aumento en el tiempo de respuesta.

Uso de librerías: las aplicaciones escritas en Java normalmente estan basadas en el uso de librerías (beans, frameworks, componentes... etc).

Una característica importante de la implementación de Jikes es, que el modelo de objeto único y la administración de memoria, permiten un chequeo

de null-pointers mediante hardware. También nos permite un acceso rápido a: los elementos de un array, campos y métodos.

2.1. Visión global de la estructura

PARTES GENERALES

JikesRVM esta dividido en cuatro partes fundamentales:

Núcleo de ejecución (Core runtime). El núcleo de ejecución contiene: la tabla de hilos, cargador de clases, el soporte para librerías, verificador... etc. Este componente es el responsable administrar todas las estructura requeridas para la ejecución de aplicaciones y para poder usar librerías.

Compiladores. Hay varios tipos de compiladores: "optimizing compiler", "baseline compiler". Esta parte es la encargada de generar el código ejecutable a partir del bytecode.

Gestores de memoria. Este es el componente responsable de asignar memoria y recoger los objetos durante la ejecución de una aplicación.

Sistema de optimización. Este es el componente responsable del "profiling" de aplicación. Lógicamente podemos utilizar el compilador para mejorar dicho "profiling".

ESTRUCTURA DE PAQUETES

Actualmente hay nueve paquetes en JikesRVM. Todas la clases de Jikes RVM se encuentran en uno de los siguientes paquetes:

com.ibm.JikesRVM. Son las clases para el núcleo, excepto las encargadas del soporte de librerías. Este paquete también contiene otras clases que no están incluidas en otros de los paquetes, como pueden ser las de los "baseline compilers".

com.ibm.JikesRVM.adaptive. Clases para el sistema de optimización.

com.ibm.JikesRVM.classloader. Este paquete contiene la implementación de las cargadores de clases y de las estructuras de datos asociadas, incluyendo la representación de las clases, métodos, etc.

com.ibm.jikesRVM.jni. Este paquete contiene la implementación del JNI.

com.ibm.jikesRVM.memoryManagers.JMTk. En este paquete se encuentran las clases en el nuevo JMTk (Java Memory Management Toolkit) conjunto de administradores de memoria.

com.ibm.jikesRVM.memoryManagers.vmInterface. Son las clases que están relacionadas con el gestor de memoria, estas clases se ocupan de la interfaz con la máquina virtual.

com.ibm.JikesRVM.opt. Son las clases relacionas con el "optimizing compiler", excepto para las clases "IR-related".

com.ibm.JikesRVM.opt.ir. Son las clases relaciones con IR ("intermediate representation" representación intermedia) del "optimizing compiler".

com.ibm.JikesRVM.osr. Clases relacionadas con el "On-Stack-Replacement".

com.ibm.JikesRVM.quick. Son las clases relacionadas con el "quick compiler".

La estructura que distribuye los directorios no sigue el convenio de Java, en el que árbol de directorios (de los archivos fuentes) esta marcado por la estructura de los paquetes, por ejemplo, no esta el directorio com/ibm/JikesRVM en alguna parte bajo $RVM_ROOT/rvm.

En cambio, la estructura de directorios fuente sigue una estructura más lógica. La "boot image builder" copia los archivos fuente desde el árbol $RVM_ROOT/rvm en el directorio. Los scripts que llevan a cabo esta copia, crean la estructura de directorios requerida por la semántica de Java y colocan las clases en los directorios apropiados.

Este acercamiento evita la necesidad de cambios en la estructura de directorios consigue la estructura de paquetes un mejor comportamiento.

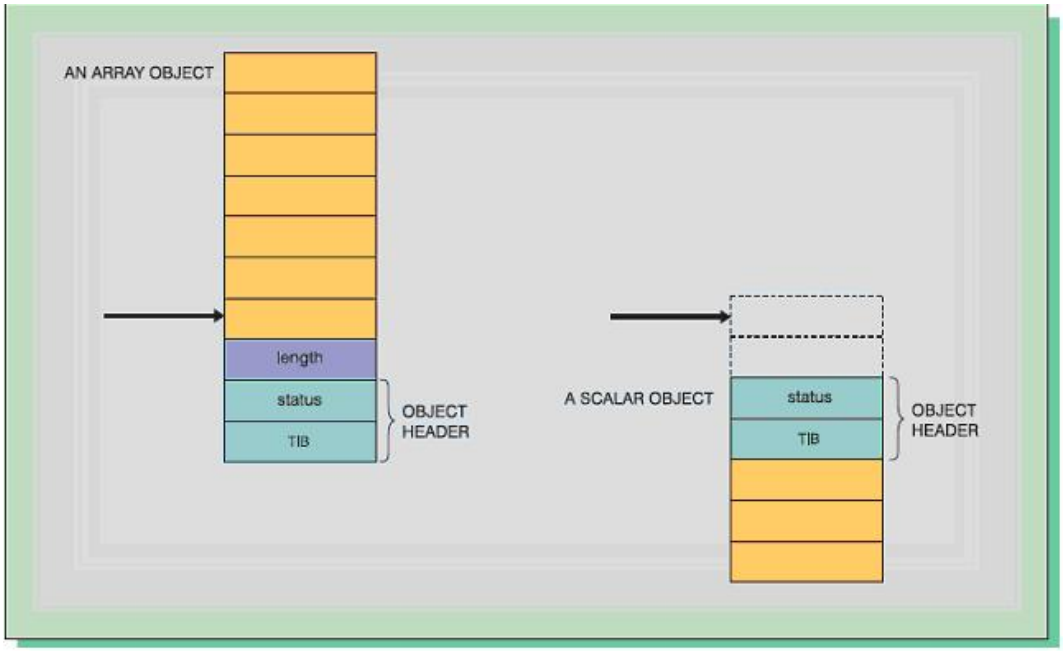
2.2. Modelo de los objetos

El modelo de objeto nos indica como van a ser representados los objetos cuando son almacenados en memoria. Un mejor modelo de objeto, nos maximizara la eficiencia con respecto a la frecuencia en las operaciones (propias del lenguaje) minimizando tambiń su tamaño en memoria.

El objeto de modelo de JikesRVM se encuentra definido en *VM_ObjectModel.java*. El modelo "JikesRVM object" ha sido sometido a grandes cambios entre las versiones del sistema 2.3.3 y 2.3.4.

En particular, en la 2.3.3 y antes del modelo de objeto "reverse scalar" el modelo consistía en: objetos escalares y arrays que se localizaban en diferentes direcciones de memoria para optimizar el chequeo de punteros nulos.

Visión general. Los valores en el lenguaje de programación de Java son primitivos (ej. int, double , etc) o son referencias (que son punteros) a objetos. Los objetos son arrays con elementos u objetos escalares con campos. Los objetos están compuestos lógicamente por dos secciones primarias: un objeto cabecera (que se describirá con mayor detalle abajo) y los campos del objeto (también pueden ser elementos del array).



Los siguientes requisitos no funcionales, rigen el modelo de objeto de JikesRVM:

- Los accesos al campo y al array serán lo más rápido posible.
- El chequeo de punteros nulos será realizado por el hardware en la medida de lo posible.
- El "method dispatch" y los otros servicios frecuentes de la ejecución serán lo más rápidos posible.
- Intentaremos minimizar el tiempo de ejecución de otras (las menos frecuentes) operaciones de Java para que no sean prohibitivamente lentas.
- Por último el tamaño en memoria (por ejemplo el tamaño de cabecera) sera lo más pequeño posible.

Asumiendo que la referencia a un objeto reside en un registro, el código compilado puede acceder a los campos de los objetos mediante un desplazamiento fijo en una instrucción simple.

Para facilitar el acceso al array, la referencia apunta al primer elemento (cero) del array y los elementos restantes son colocados en orden ascendente. El número de elementos en un array (la longitud) se encuentra justamente antes del primer elemento. De esta manera, el código compilado puede acceder a los elementos del array por medio de la base más direccionamiento indexado.

El lenguaje de programación Java requiere que un intento de acceso a un objeto mediante a una referencia a objeto nulo genere a NullPointerException. En JikesRVM, las referencias son direcciones máquina, y null esta representado por la dirección 0.

En Linux, los accesos a memoria muy baja o muy alta, pueden ser atrapados por el hardware, de esta manera todos los chequeos nulos pueden ser hechos implícitamente. Sin embargo, el sistema operativo AIX permite cargar desde memoria baja, pero los accesos a memoria muy alta normalmente causa interrupciones del hardware. Por tanto en AIX solo un conjunto de punteros pueden ser protegidos por chequeo de nulls implícito.

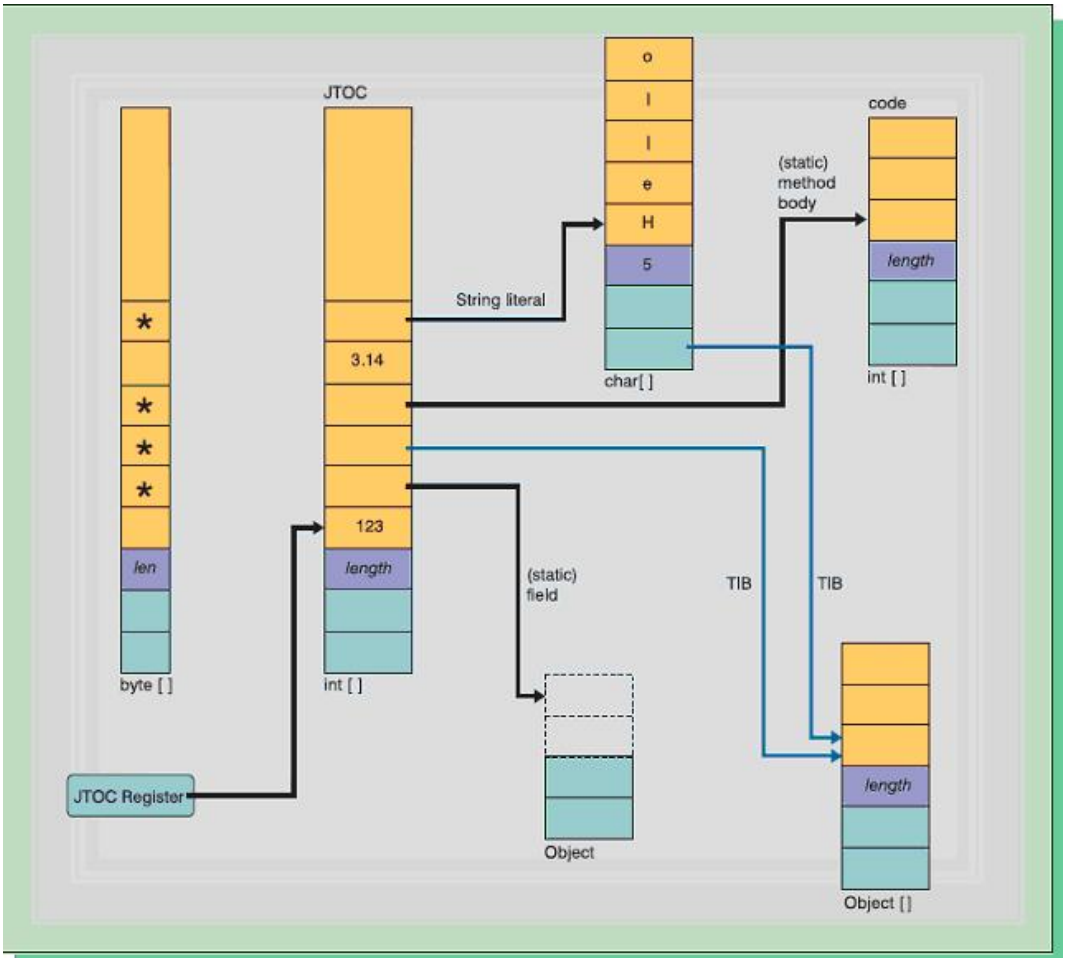
Cabecera del objeto. Lógicamente, todas las cabeceras de objetos contienen los siguientes componentes:

- **Puntero al TIB**. El TIB (Type Information Block) mantiene la información que aplicamos a todos los objetos de un tipo. La estructura del TIB esta definida por *VM_ TIBLOLayoutConstants.java*. El TIB incluye una tabla de métodos virtuales, un puntero a un objeto representante del tipo, y punteros a algunas estructuras de datos, que facilitan la eficiencia de las invocaciones del interfaz y el chequeo dinamicos de tipos.
- **Default hash code**. Cada objeto Java tiene un "Default Hash Code".
- **Bloqueo**. Cada objeto Java tiene asociado un estado de bloqueo. Puede ser un puntero a un objeto de bloqueo o una representación directa del bloqueo.
- **Longitud del array**. Todos los objetos arrays vienen provistos por un campo "longitud", que contiene el número de elementos del array.
- **Información para la recogida de basura**. Cada objeto Java tiene información asociada que usa el gestor de memoria. Usualmente esta información consiste en una o dos marcadores (bits), pero podría incluir también cierta combinación entre un contador de referencias y puntero (forwarding pointer), etc.
- **Misc fields**. En configuraciones experimentales, el objeto cabecera puede ser agrandado mediante campos adicionales para cada objeto.

Una implementación de esta cabecera abstracta esta definida por estos tres archivos: *VM_ JavaHeader* , que es la encargado del TIB, del "default hash code" y del bloqueo; *VM_ AllocatorHeader.java*, esta encargado de información de recolección de basura; y *VM_ MiscHeader.java*, que es la que proporciona campos adicionales para todos los objetos.

2.3. Métodos y campos

Un cuerpo de un método compilado es un array de instrucciones máquina. El JTOC ("Jikes RVM Table of Contents") almacena punteros a campos estáticos y a métodos. Sin embargo, los punteros a campos y métodos instanciados son almacenadas en la clase TIB. Consecuentemente, el mecanismo varia entre los mtodos estáticos y los métodos instanciados.



El JTOC mantiene punteros para cada una de las estructuras globales de JikesRVM, como pueden ser literales, constantes númericas y referencias de constantes de tipo String. El JTOC también mantiene referencias al TIB por cada clase. Desde estas estructuras puedo tener muchos tipos siendo el JTOC declarado como un array de enteros. JikesRVM usa un descriptor de array, co-indexado con el JTOC, para identificar las entradas que contienen referencias.

Métodos virtuales. El TIB contiene punteros a los cuerpos de los todos compilados (código ejecutable) para los métodos virtuales y otros a métodos instanciados de esta clase. Así de esta manera, el TIB sirve como una tabla de métodos virtuales.

Campos estáticos y métodos. Los campos estáticos y punteros a cuerpos de todos estáticos están almacenados en el JTOC. El envío de los métodos estáticos es más simple que el de los virtuales, el JTOC mantiene la dirección del cuerpo compilado de dicho método.

Métodos instanciados de inicialización. Los punteros a cuerpos de métodos instanciadosde inicialización están almacenados en el JTOC. (Estos siempre están enviados con el "invokespecial bytecode").

2.4. Carga de clases

JikesRVM implementa la carga de clases dinámicamente, que es una caracterísiteca del lenguaje de programación Java. Mientras clase esta siendo cargada puede estar en uno de los siguientes siete estados. Estos son:

- VACANT (libre). La clase está en el comienzo del proceso de carga.
- LOADED (Cargado). El archivo donde se encuentra el bytecode de la clase ha sido leído y analizado (sintácticamente) de forma satisfactoria.
- RESOLVED (Resuelto). La superclase (de la clase) ha sido cargada y resuelta y sus campos y métodos han sido calculados.
- INSTANTIATED (Instanciado). La superclase ha sido instanciada y los punteros a los métodos compilados han sido insertados en el JTOC (para métodos estáticos) y en el TIB (para métodos virtuales).
- INITIALIZING (Inicializando). La superclase ha sido inicializada y la inicialización de la clase se esta ejecutando.
- INITIALIZED (Inicializado). La superclase ha sido inicializada y la inicialización de la clase ha sido ejecutada.

A continuación vamos a detallar la transición entre los distintos estados de la carga de las clases:

- VACANT (libre). El objeto *VM_Class* para esta clase ha sido creado y almacenado, y se encuentra en el comienzo de la carga.

- LOADED (Cargado). En este estado, el archivo de la clase ha sido leído y analizado. Los métodos declarados y los campos de la clase han sido cargados. La carga de un método o de un campo consiste en leer sus atributos y modificadores. Las superclases y súperinterfaces también han sido cargadas.

- RESOLVED (Resuelto). En este estado las superclases y súperinterfaces de esta clase han sido resueltas. Una lista de los métodos virtuales y los campos instanciados de esta clase, incluyendo los métodos y campos heredados de las superclases, han sido construidos. También los desplazamientos de los campos instanciados han sido calculados.

  El espacio ha sido reservado en el JTOC para todos los campos estáticos de la clase y para los punteros a métodos estáticos, también sus respectivos valores han sido calculados. El TIB ha sido inicializado y los valores de sus métodos virtuales han sido calculados.

- INSTANTIATED (Instanciado). En este estado las superclases han sido instanciadas. Los espacios en el TIB son rellenados con punteros a código compilado o a "lazy compilation stubs" para los métodos virtuales. Los espacios del JTOC son rellenados también con punteros a código compilado o a "lazy compilation stubs" para los métodos estáticos.

- INITIALIZING (Inicializando). La superclase ha sido inicializada y la inicialización de la clase se esta ejecutando.

- INITIALIZED (Inicializado). La superclase ha sido inicializado y la inicialización de la clase ha sido ejecutada.

2.5. Magic

En este apartado hablaremos acerca de "magic", que es una ventana que JikesRVM nos proporciona para implementar funcionalidades que no son posibles utilizando el lenguaje de programación de Java puro. Por ejemplo, el recolector de basura de JikesRVM y el sistema deben, en ocasiones, acceder a memoria o hacer casts inseguros. Se desaconseja encarecidamente el uso de magic y de su código, excepto cuando sea absolutamente necesario.

Hay 3 tipos de operaciones en magic, las cuales describiremos a continuación:

- La primera es una colección de "métodos mágicos" que son los métodos estáticos de la clase *VM_Magic*.

- La segunda, son las clases mágicas *VM_Address*, *VM_Word*, *VM_Offset* y *VM_Extend*, que se usan en la parte de ejecución y recolección de basura.

- La tercera, son varios mecanismos para declarar "código ininterruptible".

La clase VM_Magic. Ciertos todos en la clases *VM_Magic* son tratados de forma distinta por el compilador. Las razones de esta distinción son que: estos métodos hacen acceden a memoria o a otros estados de la máquina, realizan casts inseguros, o ejecutan otras llamadas al sistema operativo que no han sido implementadas en el código de Java.

Los desarrolladores tienen que ser extremadamente cuidadosos a la hora de escribir código que use la clase *VM_Magic*. El uso de *VM_Magic.objectAsAddress* para tratar los punteros aritmticos es especialmente complicado, dado que podemos perder estos punteros durante la recogida de basura. Todas las funciones de "magic" tienen que realizarse durante los "métodos interruptibles".

Dado que magic es inexpresable en el lenguaje de programación Java, no nos sorprende que los cuerpos de los métodos de *VM_Magic* no estén definidos. Para cada uno de estos métodos, las instrucciones Java que generan el código se encuentran almacenadas en *OPT_GenerateMagic* y *OPT_GenerateMachineSpecificMagic*, también interviene *VM_Compiler* (a la hora de generar el código ensamblador). Siempre que el compilador se encuentra una llamada a uno de estos métodos mágicos, él introduce "inlines" el código apropiado (para el método magic) dentro de la llamada al método.

2.6. Implementación del JNI

A continuación describiremos como interactua JikesRVM con el código nativo. Los tres aspectos más importantes son:

Las funciones JNI. Este es el mecanismo que se encarga de la de transición del código nativo a código Java. JikesRVM implementa estas funciones a través de las especificaciones del JNIEnv.

Las funciones se encuentran definidas en la clase *VM_JNIFunctions*. Tenemos que tener en cuenta, que estas funciones están compiladas con prólogos y epílogogs especiales, que traducen los convenios de las llamadas nativas a convenios de las llamadas Java manteniendo otros detalles de la transición. Actualmente el "optimizing compiler" no soporta estas secuencias de prólogos y epílogos, por lo tanto todos los métodos en esta clase están compilados mediante el "baseline compiler".

Las secuencias de prólogos y epílogos están generadas actualmente por una plataforma específica, el *VM_JNICompiler*.

Métodos nativos. Son el mecanismo que se encarga de la transición del código Java a código nativo. Para invocar los métodos nativos podemos utilizar dos tipos de mecanismos. El primero es el mecanismo normal, que JikesRVM usa para invocación de estos métodos normalmente. El segundo, es un mecanismo de llamadas más restrictivo, que se usa de forma interna (por el código de bajo nivel de JikesRVM). A continuación detallaremos ambos mecanismos:

- *Invocación de los métodos nativos*. El primer mecanismo, es cuando JikesRVM realiza esta transición mediante llamadas a métodos de la clase *VM_Syscall*. Estos métodos nativos son definidos e invocados en uno de los archivos (en particular son los de C y C++) del ejecutable de JikesRVM. Para poder implementar una llamada al sistema, los compiladores de RVM generan una secuencia de llamadas, estas tienen que ser las adecuadas para el convenio de la plataforma. Hay que tener claro que los métodos de las llamadas a sistema no son métodos JNI.

  El segundo mecanismo es el JNI, este procedimiento es el más frecuente a la hora de invocar métodos nativos. Para poder escribir código JNI lógicamente deberemos de usar el interfaz JNI. Cuando tengamos que implementar una llamada a un método nativo, JikesRVM usa *VM_JNICompiler* que a su vez genera una rutina que administra la transición entre el código Java y el código nativo.

- *Integración con "m-to-n threading"*. El mayor cambio sufrido durante el proceso de mejora de JikesRVM, ha sido que el modelo "m-to-n threading" funcione correctamente con el código nativo. En este proceso de mejora, se ha tenido que rediseñar tanto el JNI como el sistema de hilos (aún se sigue trabajando en ello).

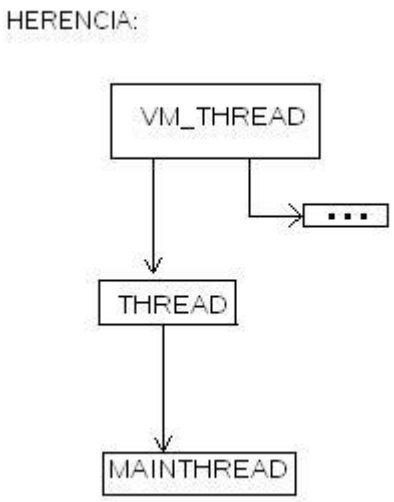
3. Detalles del gestor de hilos

3.1. Sistema de hilos

La máquina de JikesRVM tiene un sistema complejo para administrar y sincronizar sus hilos, en un principio, se intentará dar una visión general que se explicará a continuación.

JikesRVM tiene una clase principal para los hilos llamada *VM_Thread*, todos los hilos creados durante la ejecución y compilación de la máquina derivan de esta clase principal. Las clases de hilos que heredan directamente son: *DebuggerThread*, *FinalizerThread*, *OSR_OrganizerThread*, *VM_CollectorThread*, *VM_CompilationThread*, *VM_ControllerThread*, *VM_IdleThread*, *VM_Organizer*. Todos los demás hilos de la máquina también heredan de esta clase pero de una forma indirecta.

Una cosa a tener en cuenta es que además de los hilos que usa la máquina virtual también heredan de ella la clase *Thread.java*. Ésta se encuentra en la librería de Java[1] que utiliza JikesRVM, lo que significa que los cambios que se efectúen en esta clase también afectarán a la ejecución de hilos de la aplicación que se quiera ejecutar, ya que esos hilos se rigen por la clase de Java.



Dependiendo del número de procesadores los hilos principales se multiplican para que todos ellos estén en todos los procesadores y así manejar correctamente el funcionamiento de la máquina. La máquina puede tener

[1] libSupport/java/lang

uno o más procesadores virtuales, gracias a la clase *VM_Processor*, pero si no se le especifica directamente cuantos procesadores se quiere que se utilice, solo creará uno por defecto, en nuestro caso siempre hemos trabajado con un sólo procesador, por tanto sólo habrá una copia de los hilos principales.

Hay que comentar también que existen ciertas estructuras para gestionar estos hilos, ya que, el hecho de que estén creados no significa que estén todos activos, y por supuesto se tendrán que ceder el control de la aplicación constantemente unos a otros, todo esto lo controla el planificador[2], con la ayuda de una serie de colas y arrays donde almacena los hilos.

La clase principal de las colas[3] es una clase abstracta *VM_AbstractThreadQueue.java* que implementa los todos clásicos de una cola. En esta estructura se encolarán *VM_Threads*, y por tanto como hemos dicho todos los hilos de la máquina. Esta clase la implementan cuatro clases directamente y muchas otras clases más indirectamente.

Estas colas se diferencian entre ellas por el tipo de hilos que almacenan o el estado de estos, por ejemplo si están esperando a un evento o están listos a ser despertados, si están esperando a un evento en general o a un evento de un proceso. Así el planificador consultará una cola u otra dependiendo de sus necesidades.

El comportamiento general de estas colas se puede diferenciar entre la global la cual la manejan todos los procesadores activos y las locales de cada procesador, las globales requieren de sincronización para controlar el acceso simultáneo a ellas, para ello utiliza los procesos *VM_ProcessorLocks*. En este tema no vamos a profundizar ya que al solo trabajar con un procesador no hemos tenido necesidad de manejarlo, y por tanto tampoco de conocer bien su funcionamiento.

Para realizar la transferencia del control de la ejecución entre un hilo y otro, para acceder a un objeto común, se utilizan los métodos *yield* y *morph*, estos métodos pertenecen a la clase *VM_Theard* y el método *dispatch* de la clase *VM_Procesor*. El método *yield* busca la cola adecuada para encolar este hilo, luego *morph* hace algunas operaciones y cede el control a *dispatch* que se encarga de suspender el primer hilo y despertar el hilo que considera debe ser despertado para ser ejecutado. Entonces *dispatch* invoca al método *threadSwitch* de la clase *VM_Magic*. Este método trabaja a más bajo nivel, conectando con el hardware, lo que hace es restaurar el entorno del segundo hilo, salvando previamente el entorno del que deja de tener el control.

Para evitar que este hilo sea intentado ejecutar en *yield* o *morph* mientras se encuentra en proceso de salvar su entorno, se utiliza un atributo que se

[2]Scheduler
[3]Queue

activa y así no se utiliza. Al igual que este atributo, los hilos, tienen muchos otros flags que ayudan a su manejo, dan información como el estado que se encuentran por ejemplo. Así que se basara en esos flags el método *dispatch* para elegir el hilo que va a ser ejecutado, ya que JikesRVM no tiene ningún sistema de prioridad de hilos. Esto significa que cualquier hilo que esté listo para la ejecución puede ser elegido.

Jikes, con las clases *VM_LocK* y *VM_ThinLock*, coordina la sincronización entre los hilos, para sincronizarlos utiliza el bloqueo de objetos para controlar el acceso a ellos. Para ello utiliza un esquema de bloqueo de peso ligero[4]. El sistema consiste en que cuando un objeto es llamado se le aplica un bloqueo de peso ligero, esto se realiza indicándolo en unos bits que hay en la cabecera del objeto para este cometido. Si este bit ya estuviera activado se convierte en un bloqueo de "peso pesado", y los bits de la cabecera serán el índice de un array de objetos bloqueados.

De todas formas, si el objeto está bloqueado, tendrá información sobre el identificador del hilo que lo ha bloqueado (el que lo esta utilizando y en ese momento se está ejecutando en él) y cuantas veces ha sido solicitado y por tanto las veces que va a ser utilizado. Si un hilo intentara acceder directamente a un objeto que ya está bloqueado, bloquearlo daría problemas. Por tanto siguiendo este sistema los objetos que están bloqueados con un bloqueo pesado tiene una cola para hilos que estén intentando conseguir acceder a este objeto.

Se utiliza un mecanismo similar para solucionar el problema de los hilos con bloqueos de "peso pesado" que estén esperando y despertarles para que continúen su ejecución. Todos estos bloqueos tiene una cola una "waitingQueue" en la que están los hilos que han sido bloqueados y están esperando en un "wait". Cuando se les despierta mediante un *notify* el hilo se le desencola y se le encola en otra para hilos preparados para ser despertados, una *wakeupQueue*. El orden de estas colas, se utiliza a modo de prioridad a la hora de elegir al que hay que despertar, ya que Jikes no tiene prioridades.

Por lógica un hilo podría estar en las dos colas de espera simultáneamente, pero debido a la implementación no se puede. Para controlar los tiempos de espera es algo necesario, así que para solucionarlo, se crean dos nuevas colas, iguales que las que tratábamos pero que en lugar de encolar hilos lo que almacena son "proxies", objetos de la clase *VM_Proxy*. Este objeto representa hilos y puede representarlo en más de una cola de espera. Sólo tiene que cumplir dos condiciones para un correcto funcionamiento, que no se borren de las dos colas a la vez y que es en las dos colas, ya que sólo se almacenan si es necesario tenerlo en las dos. La sincronización se encarga que

[4]light-weight locking

eso se cumpla.

3.2. Bloqueos

Debido a la importancia que tiene en los hilos el control de la sincronización y los bloqueos vamos a dedicarle una sección para profundizar algo más en ellos.

La máquina utiliza *VM_Processor* y *VM_Lock* como sustitución al manejo con hilos de monitores que se puede implementar en Java. Este sistema no es eficiente si el sistema de monitores es suficiente, es adecuado para situaciones más complejas en las que los objetos a los que se va a intentar acceder son varios.

El objeto *VM_Processor* reduce un montón de hilos de *VM_Thread* a unos pocos hilos del kernel. A estos hilos generados se les llama Posix threads. A la máquina no le importa si están implementados como hilos del kernel o como hilos a nivel de usuario. Sólo le importa cuantos hilos de este tipo se le especifica que tenga, estos hilos serán como procesadores virtuales, por tanto se puede especificar cuantos existen como antes se ha indicado. Lo adecuado es que haya tantos como procesadores físicos tiene la máquina.

La clase *VM_Lock* , puede descomponerse en cuatro secciones, que serían las operaciones que realiza:

- Apoyo a la sincronización de objetos. Esto implica que cuando un objeto se le aplica las instrucciones *wait*, *notifyAll*...de Java, esta clase se encarga de que el objeto sea bloqueado o desbloqueado dependiendo de la instrucción por el hilo que los este ejecutando en ese momento.

- Realiza en los objetos bloqueados el bloqueo de "peso pesado" y guarda en ellos la información que hemos dicho anteriormente que contiene. Este mecanismo se puede dividir en dos subsecciones:

  - La primera parte del mecanismo se encarga de bloquear y desbloquear los objetos que esén asociado con un bloqueo pesado.
  - La segunda parte se encarga de asociar y disociar los objetos bloqueados con los bloqueos pesados.

- Se encarga de distribuir y mantener los bloqueos de "peso pesado". Esto quiere decir que se encarga de reservar memoria y liberarla para estos bloqueos. También distribuye los bloqueos entre los distintos procesadores si los hubiera.

- Depuración y apoyo a la puesta apunto del rendimiento. Lo que hace está sección es dados unos problemas que se suelen dar les aplica la solución adecuada. Algunos ejemplos de tareas de este punto que todavía no han sido solucionadas adecuadamente son:
  - ¿Qué hacer si en el intento de bloquear un objeto falla?
  - ¿Cuántos bloqueos pesados son necesarios?, ¿si son muchos como trabajar con ellos?

Esta clase respetaría los siguientes requisitos, que sea posible bloquear un objeto que tiene prohibida la reserva de memoria y que una vez este bloqueado un objeto ningún hilo pueda acceder a él. Estas dos condiciones es esencial que se cumplan para poder asegurar una correcta sincronización.

Estas clases manejan dos tipos de bloqueos: el bloqueo del procesador (*VM_ ProcessorLock*) y un bloque más bil (*VM_ ThinLock*). El *ProcessorLock* es más sencillo, permite bloquear desbloquear, y algún método más, es sobre *ThinLock* sobre el que se puede pasar de bloqueos ligeros a pesados.

3.3. Tiempo de ejecución

La máquina virtual una vez se este ejecutando tiene varios hilos que crea y gestiona. Cada uno de ellos tiene un cometido distinto y todos actúan de forma independiente. Estos hilos son:

- Principal
- Recolector de basura
- Finalizador
- Depurador
- Planificador
- ...

También hay que decir que en tiempo de compilación se crea un hilo que dirige la compilación, esto es porque la máquina se compila a si misma.

Vamos a hablar de algunos de ellos ya que son muchos y de algunas de sus características.

El *mainThread* es el hilo principal de la ejecución de las aplicaciones, realiza todas la operaciones indicadas en el código de la aplicación.

El *DebuggerThread* simplemente se utiliza para depurar la máquina en caso de error para corregir errores. Se activa mediante el todo *threadSwitch()*

El *FinalizerThread* es invocado en el *boot* del planificador, trabaja con el recolector de basuras. Produce la cola de espera para el recolector de basuras, en todos los métodos que se tengan que finalizar les ejecuta su método *finalize()*. Este hilo estará esperando hasta que el recolector de basuras lo despierte.

VM_ CollectorThread participa en la recolección de basura, por cada procesador (*VM_ Procesor*) hay un hilo recolector de basura , como trabajamos sobre un procesador solo tenemos un hilo. Realiza los siguientes pasos:

1. Espera que se le solicite para recoger basura.
2. Se sincroniza con los otros recolectores, en nuestro caso no hay otros colectores.
3. Recupera espacio en memoria.
4. Se vuelve a sincronizar antes de acabar y vuelve al paso uno.

El hilo que se crea en compilación, el *VM_ CompilationThread*, tiene una cola de objetos para ser compilados y escoge el que tiene más prioridad para que se comile el primero. Todas las decisiones que toma son dirigidas por el *cotrollerThread*.

ControllerTheard es el cerebro de la optimización del sistema, recibe información de las medidas en tiempo de ejecución del sistema. Tambien influye en el hilo de compilación para que optimice los ejecutables.

3.4. Modificaciones realizadas

Podemos dividir las modificaciones en dos partes, una sería las modificaciones en el uso de los hilos, y la otra sería las modificaciones en la estructura de los hilos.

La primera en realizarse sería la que se realiza con el uso de los mismos. El planteamiento es el siguiente, pasamos de tener una aplicación a tener dos, por tanto, si con una aplicación se creaba un hilo principal de ejecución, a tener dos necesitaríamos duplicarlo, independientemente de lo que cada hilo haga después, en un principio deberían tener las mismas características.

Por tanto nos situamos en la clase principal de Jikes RVM, esta se encuentra en el directoria raíz y es *VM.java*. En esta clase se prepara la máquina para su ejecución tanto en compilación como en ejecución. Prepara todas las clases, controla el recolector de basura, inicia la ejecución. La mayoría de sus

métodos son estáticos para que puedan ser accedidos desde cualquier punto de la máquina virtual.

Una de las cosas que hace es comenzar la ejecución de la aplicación. Para ello da los siguientes pasos:

- Recoger como argumentos en un vector de String de la máquina la información dada por el usuario para ejecutar la aplicación.
- Inicializa los conectores y ejecuta el controlador.
- Halla la clase que le han indicado ejecutar en el primer argumento y la busca en el directorio donde se encuentra.
- Crea el hilo principal[5].
- Inicia el hilo principal.
- Crea el hilo para depurar.
- Y termina el hilo que controla el inicio de la ejecución.

El usuario cuando realiza la llamada a la máquina le especifica la aplicación a ejecutar y sus argumentos. Como ahora se trata de dos aplicaciones y sus argumentos, se separaran mediante una arroba '@'. El sistema recibe esos argumentos en un array de strings. Éste método distinguirá y almacenará la aplicación primera y sus argumentos en un array y los que aparezcan después del símbolo como aplicación y argumentos de la segunda. Una vez separados se trata la primera aplicación y luego la variable del sistema toma el valor del segundo vector que contiene los datos de la segunda aplicación.

El siguiente paso será la duplicación de hilos con sus valores. Se comprueba que exista una segunda aplicación, para que no ocasione ningún problema. En el supuesto de que se haya utilizado como mono-tarea, que a pesar de ser menos eficiente, ya que se desaprovechan recursos, es una opción válida, así que no daría ningún problema. Una vez comprobado, se crea un hilo principal para la aplicación, se inicializa al igual que se hace con el primero, pero con sus parámetros específicos, y se inicia.

Con únicamente estos pasos, la máquina ya sería multi-aplicación, ya que habría un hilo de ejecución para cada una, de hecho se pueden ejecutar sin problemas dos aplicaciones. El único problema sería que al no haber duplicado los recursos de memoria y acceder los dos a los mismos, una aplicación podría interferir en la otra.

[5] mainThread

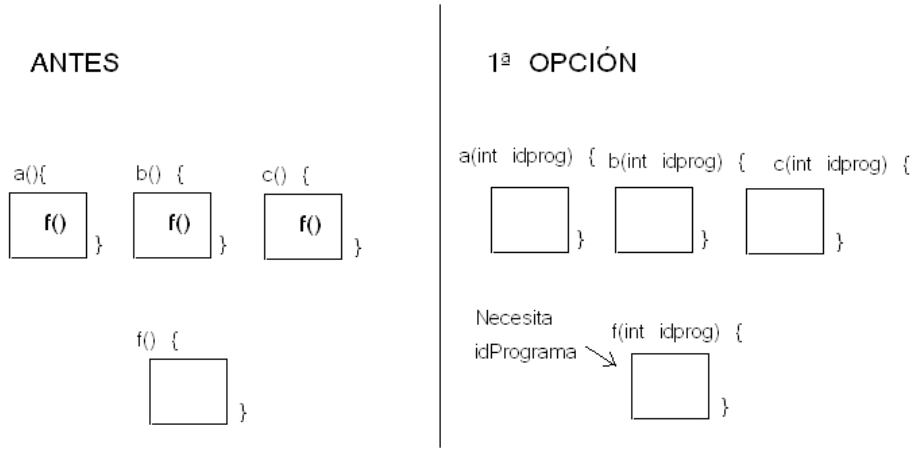
El siguiente problema que nos encontramos era descubrir la manera de diferenciar entre el hilo de una aplicación y otra, para ello le añadimos a *VM_ Thread* un atributo *idPrograma* con su correspondiente accesor. Al añadirlo a *VM_ Thread* todos los hilos tendrán este atributo ya que heredan de él.

También heredaría este atributo la clase de java *Thread.java*, pero no se pueden añadir modificaciones en ella ya que esta es la clase que utiliza el compilador para compilar y ejecutar las clases de las aplicaciones, por tanto cualquier modificación en esta clase afecta directamente a las aplicaciones que usan JikesRVM. Los hilos que cree la aplicación dentro de su código serán de este tipo, con el consiguiente problema a la hora de asignarle identificador. Se solucionó asignándole el mismo identificador que el hilo activo en ese momento, esto es porque el hilo activo es el de la aplicación que le llama, es decir si la aplicación uno es la multihilo, le asignará un uno si es la segunda será un dos.

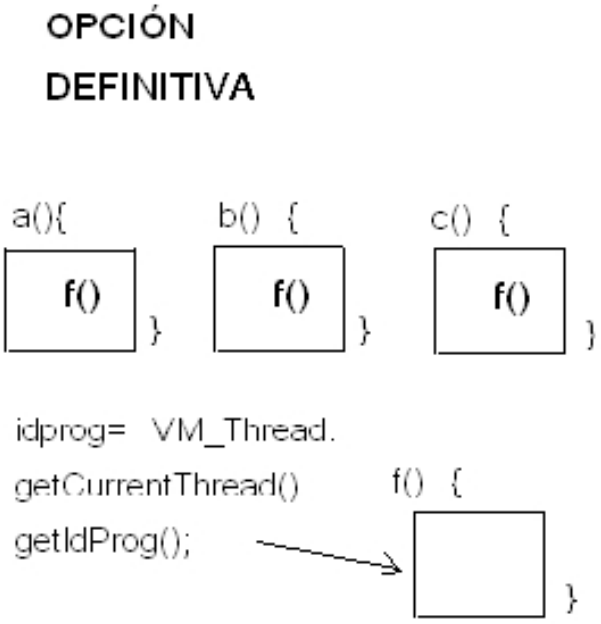
El resto de las clases que heredan de hilos también tiene constructoras con el identificador de programa como parámetro para que se les pueda asignar en el momento de construirlo, y poder así llamar a la constructora de su clase padre. Aunque el único hilo que se le asigna directamente el identificador de programa, sin consultar el identificador de ningún otro hilo, será el del hilo principal de la aplicación.

De esta forma se pueden distinguir los hilos de una aplicación y otra, todos los demás hilos de la aplicación, el principal de la ejecución, el que se utiliza para depurar, el del recolector de basuras... todos ellos tendrán el identificador con valor uno, así que si necesitan acceder a memoria lo harán en la parte reservada para la aplicación uno.

En un principio para realizar esta mejora, la planteamos como una propagación del parámetro idprog, desde el punto que es necesitado, hasta algún punto de la máquina en el que se pudiera asegurar su valor. El punto inicial se encuentra dentro del planificador la clase Plan.java, que es donde se tratan los recursos del heap, para saber dependiendo de que hilo está accediendo a los recursos proporcionarle unos u otros. Desde ese punto se les pasaba por parámetro el valor del identificador del programa que lo necesitaba, ya que supusimos en principio que el hilo que estaría ejecutando esa clase sería el del planificador y por tanto de valor uno. De esta forma propagábamos el valor, de un todo a otro. El problema de esta solución además de tratarse de un trabajo muy costoso fue que al llegar a uno de los últimos métodos que lo necesitaba que para propagarlo había que acceder a clases de la librería de Java, y que por tanto no debíamos modificar.



Luego al buscar otra solución alternativa nos dimos cuenta que el hilo que accedía al objeto Plan, es directamente el de la aplicación que necesita los recursos, al ser el método que da el hilo que está en ejecución en ese momento estático no hay problemas para acceder a él, por tanto cada vez que se va a acceder a un espacio de memoria se consulta el identificador del programa al que pertenece, al realizarlo de esta manera cada aplicación puede tener un número indefinido de hilos, sin afectar al buen funcionamiento de la máquina, porque todos los hilos de la misma aplicación tendrán el mismo identificador.



Otro problema que nos surgió es que el recolector de basura al tener identificador uno, sólo visitaba los espacios de memoria destinados a la aplicación uno, con el consiguiente problema de memoria en el resto de los espacios usados. Duplicar el hilo del recolector de basura para cada aplicación no es eficiente, habría que duplicar también toda las estructuras que acompañan y le son necesarias al recolector, por tanto se duplicarían recursos que no son necesarios tener diferenciados e incrementaría el espacio de memoria.

La solución que aplicamos a este problema es comprobar primero si el espacio que necesita que el recolector de basura pase por él es de la aplicación uno, si no fuera así se generaría una excepción la cual capturaríamos, porque esto no significa que sea un error, entonces si la excepción es capturada se aplicaría la misma operación sobre el espacio de la aplicación dos, resolviendo así el problema. En el supuesto que aún así se tuviera que generar una excepción porque ninguno de los dos espacios es el esperado, debido a algún error del sistema Jikes RVM, se generaría la excepción en el espacio de la aplicación uno, se capturaría, entonces se intentaría acceder al espacio de la aplicación dos, y generaría otra excepción la cual no estará capturada y se pararía la ejecución como es lo esperado ante un error irrecuperable del sistema.

Hemos hecho que escriba por pantalla cada vez que crea un hilo el identificador que le asigna. Si ejecutamos dos aplicaciones sencillas en el que la primera realiza unos cálculos y la segunda hace unas operaciones con un hilo que genera esta misma aplicación el resultado es el siguiente:

```
Hilo creado para el programa 1
Hilo creado para el programa 1
Hilo creado para el programa 1
Hilo creado para el programa 1
Hilo creado para el programa 2
Hilo creado para el programa 1
Hilo creado para el programa 2
```

Los tres primeros hilos son de ejecución, el cuarto es el de la primera aplicación, el quinto es el de la segunda aplicación, el sexto es otro hilo del sistema, el hilo de depuración, y el séptimo es el hilo que crea el hilo principal de la segunda aplicación dentro.

Con estos cambios y los indicados en el apartado *Detalles del gestor de memoria (Modificaciones)*, serían los suficientes para que la máquina virtual Jikes se convierta en una máquina multiaplicación que funcione correctamente con todo tipo de aplicaciones.

4. Detalles del gestor de memoria

La siguiente sección se centra en la estructura e implementación del JMTk, el gestor de memoria de la máquina virtual Jikes.

4.1. Estructura

JikesRVM está diseñada para soportar el intercambio de distintos gestores de memoria ya implementados. Cada gestor está constituido por un asignador[6] concurrente de memoria para objetos y un recolector de basura stop-the-world[7], paralelo.

Asignador concurrente de memoria para objetos. Los gestores de memoria particionan el montículo en dos espacios, uno para objetos grandes y otro para objetos pequeños. Cada gestor de memoria usa un espacio de objetos grandes que se forman mediante una estructura de bloques enlazada. En el caso del espacio para objetos pequeños existen dos enfoques distintos, y cada gestor usa uno de ellos. En el caso del gestor utilizado para este proyecto, la asignación de memoria para objetos pequeños se asimila a la primera de las que vamos a hablar, pero con algunas características añadidas. De ello se hablará en la sección sobre la implementación.

Para soportar la asignación concurrente de memoria para los objetos pequeños en el primer caso, cada procesador virtual mantiene una parte grande del espacio local en el cuál los objetos pueden ser destinados sin requerir sincronización. Estas fracciones locales no están lógicamente separadas del montículo global, por lo que un objeto alojado por un procesador virtual es accesible por cualquier otro procesador si este tiene una referencia al objeto. La asignación de memoria se realiza incrementando un puntero dentro del espacio a partir del tamaño requerido, y comparando el resultado con el límite de la porción de memoria local. Si la comparación falla, el asignador de memoria obtiene una nueva porción local de memoria del espacio global.

Sin embargo, en el otro caso, el espacio de objetos pequeños es dividido en bloques de tamaño fijo. Cada bloque es, además, subdividido dinámicamente en ranuras también de tamaño fijo. Cuando el asignador de memoria recibe una petición de espacio, este determina el tamaño de la menor ranura que puede satisfacer la petición, y obtiene el bloque actual para ese tamaño. Si el bloque actual está lleno, este está enlazado con el siguiente bloque para ese tamaño con espacio disponible, y este último se convierte en el bloque actual.

[6] *Concurrent object allocator*

[7] Implementa la funcionalidad básica de *stop the world GC*

De la mutación a la recolección. Cada procesador virtual tiene asociado un hilo recolector. JikesRVM opera en uno u otro modo: los hilos normales están ejecutándose y los hilos recolectores están inactivos, o los hilos normales están inactivos y los hilos recolectores están ejecutándose. El recolector de basura es activado cuando uno de los hilos normales lo pide, o cuando una petición de espacio por parte del hilo al asignador de memoria no se ha cumplido satisfactoriamente.

Recolector de basura paralelo. En JikesRVM los recolectores de basura han sido diseñados para poder ejecutarse en paralelo. Los hilos recolectores se sincronizan entre ellos al final de cada fase de las tres que hay. Dependiendo del enfoque que tenga el gestor, entre los dos ya comentados, la primera y última fase cambian de mecanismo.

En la fase *inicial*, el hilo recolector con el primer enfoque copia el propio objeto hilo y su objeto procesador virtual. Esto asegura que la actualizaciones de estos objetos se realizan sobre una nueva copia y no una antigua, la cual será descartada al final de la recolección. En el otro enfoque el gestor asocia a cada bloque de memoria un array, con una entrada al array por cada ranura. Durante la inicialización, todas las entradas del array están a cero. Todos los hilos recolectores participan en la inicialización.

En la fase de *identificación de raíz y escaneo*, todos los recolectores se comportan igual. Los hilos recolectores compiten por el JTOC y por cada pila de los hilos normales, escaneandolos en paralelo para encontrar raices (esto es, las referencias a objetos que se suponen fuera del montículo), las cuales son marcadas y llevadas a una cola de trabajo. Entonces los objetos accesibles a través de dicha cola son marcados también. Como parte del marcado de un objeto, si el gestor implementa el primer enfoque, copiara sus bits dentro de un nuevo espacio.

Un objeto eliminado de la cola de trabajo es escaneado para encontrar referencias a objetos. Los deplazamientos de dichas referencias son obtenidos de la clase objeto, que es la primera entrada en el TIB del objeto. Por cada una de las referencias, el recolector intenta marcar un objeto. Si se consigue, este es añadido a la cola de trabajo.

En la fase *final*, los recolectores con el primer enfoque simplemente invierten las porciones de memoria ocupada y libre del heap. Los hilos recolectores obtienen partes locales de memoria de la ahora parte vacía. Un hilo recolector con el segundo enfoque realiza los siguientes pasos:

- Si este era una recolección menor, marca todos los viejos objetos como vivos.
- Escanea todas las marcas para encontrar bloques libres, y retorna la lista de bloques libres.

- Para todos los bloques no libres, cambia el array: las entradas no marcadas en el viejo array identifican las ranuras libres para ser utilizadas en la asignación de memoria.

4.2. Implementación

En esta sección veremos como es implementado todo lo explicado anteriormente, paso previo a la sección siguiente donde veremos las modificaciones añadidas al gestor de memoria. El primer punto a ver es la estructura de la implementación. Dentro de todas las clases que componen el gestor de memoria, estás están divididas en tres grupos, que se corresponden con los directorios donde se encuentra su código. Primero las clases que definen los planes del gestor de memoria, contenidas en el directorio *plan*. Un plan especifica una configuración particular de los componentes del gestor, los cuales juntos definen la estrategia del gestor de memoria. Las clases que implementan varias políticas de gestión de memoria se encuentran en el directorio *policy*. Las clases que implementan utilidades genéricas están contenidas en el directorio *utility*.

Dependiendo de tus propósitos, puedes construir la máquina virtual con uno de los distintos planes que están implementados. Para nuestro proyecto hemos elegido el plan CopyMS. Este plan es un híbrido entre un recolector con un espacio de copia (similar al primer enfoque visto en la sección anterior), espacio usado por el asignador de memoria, y un espacio de no copia (similar al segundo enfoque), al cuál van los elementos que han sobrevivido a una recolección de basura.

Todos los planes heredan de *BasePlan*, y todos los planes son recolectores stop-the-world, por lo que heredan también de *StopTheWorldGC*, el cual implementa la funcionalidad stop the world GC. Todos los planes soportan asiganción paralela de memoria y recolección. Las funciones básicas de cada plan incluyen:

- Identificación con una capa de memoria virtual, usando *VMResource* para, por ejemplo, unir un espacio "nursery" con un rango de direcciones.
- Proporcionar asignación de memoria uniendo asignadores con los diferentes recursos de memoria (VMResources).
- Invocando la recolección de basura cuando llega el caso en que es necesario.
- Aplicando las correspondientes políticas a los objetos encontrados durante el proceso de recolección.

Implementación del plan CopyMS. Como todos los gestores de memoria implementados para JikesRVM, CopyMS trabaja con un espacio reservados para objetos grandes, considerados como tales cuando su tamaño es mayor que el máximo tamaño de bloque disponible en el espacio MS. Este espacio se llama LOS[8].

Para la asignación de memoria para objetos pequeños no utiliza solo un espacio y con uno de los enfoques vistos en apartados anteriores, sino que utiliza dos espacios, cada uno lo trata con uno de los dos modos. Trabaja con el primer modo sobre el espacio "nursery". En el son alojados los objetos recién creados. Aquí los objetos se colocan de forma contigua. Cuando se produce una recolección, lo supervivientes se mueven al espacio MS. El asignador de memoria para un objeto recién creado trabaja directamente sobre este espacio de memoria.

El segundo es el espacio MS. En el están almacenados los objetos que han sobrevivido a una recolección. La asignación se lleva a cabo mediante listas enlazadas con bloques de distinto tamaño. De aquí los objetos ya no se mueven.

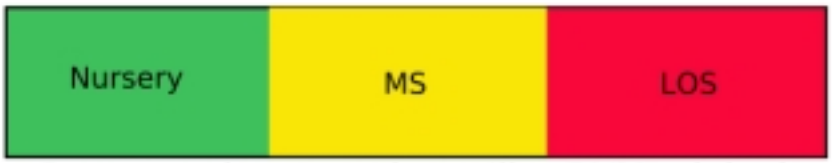
4.3. Modificaciones realizadas

Tras conseguir que la máquina virtual pueda ejecutar más de una aplicación, aún queda el problema de que ambas aplicaciones utilizan el mismo espacio de memoria para alojar los objetos.

Una vez que entendimos la estructura del gestor de memoria, explicada en los apartados anteriores, concluimos que lo necesario para conseguir nuestro objetivo era duplicar los espacios de memoria utilizadas por el plan CopyMS. Por esto, lo primero que hicimos fue crear, por cada aplicación que puede ejecutar la nueva máquina virtual, un espacio "nursery", para los objetos pequeños recién creados, un espacio MS, para los objetos que han sobrevivido a una recolección y un espacio LOS, para los objetos grandes.

Estas duplicaciones conllevaron la duplicación de los recursos de memoria utilizados por los espacios, tales como los recursos virtuales de memoria, implementados por la clase *VMResource*. El tamaño de estos nuevos recursos, en vez de ser del tamaño de los originales, tienen la mitad de tamaño. El resultado se puede apreciar en la siguiente figura.

[8]Large Object Space



Recursos de memoria en JikesRVM original

Recursos de memoria en JikesRVM multiaplicación

Pero con la separación de recursos no es suficiente. Hasta ahora los dos programas acceden y alojan sus objetos en los tres primeros espacios. De alguna manera el gestor tiene que saber a que recurso acceder dependiendo de la aplicación que esté operando. Para conseguir esto recurrimos a los cambios hechos en la parte del sistema de hilos.

Cada vez que se accede, por ejemplo, a alojar en memoria un objeto, esta petición ha sido expedida por el hilo actualmente en ejecución. Entonces la información necesaria en este caso es el identificador de la aplicación a la que pertenece este hilo. Para conseguirla necesitamos poder conocer dentro del gestor de memoria a dicho hilo. Hemos implementado esta posibilidad en todo el plan CopyMS. En toda función en la que se necesite elegir un espacio de memoria primero comprobamos el identificador de aplicación del hilo en ejecución, con el cuál decidimos si utilizar los espacios de memoria de la primera aplicación, o en el otro caso, los espacios de la segunda aplicación.

Pero con las modificaciones hasta ahora hechas no era suficiente. Aún nos encontramos con otro problema. ¿Que ocurría si el hilo que accedía a los espacios de memoria no era de ninguna de las dos aplicaciones en ejecución? Esto pasaba con el hilo recolector del procesador virtual. Por no pertenecer a ninguna de las aplicaciones solo accedía a los espacios de memoria originales. El problema que ocasionaba esto era cuando intentaba acceder a un objeto de la segunda aplicación a través de una referencia a los nuevos espacios de memoria. Esta referencia causaba una violación en el espacio de memoria original, con su correspondiente excepción.

Para solucionar el problema dejamos que el hilo recolector accediera a todos los espacios de memoria. Primero accede al espacio de memoria de la primera aplicación y después al de la segunda aplicación. Cuando el recolec-

tor utiliza una referencia de un objeto de la segunda aplicación, este intenta buscarlo en el espacio de memoria de la primera aplicación. Al no encontrarlo causa una violación en el espacio y lanza una excepción. Gracias a esta excepción sabemos que ha fallado la búsqueda, la capturamos y permitimos al hilo recolector buscar el objeto en el espacio de memoria de la segunda aplicación. Esta última búsqueda se ejecuta correctamente.

Con todos estos cambios hemos conseguido que cada aplicación tenga sus propios espacios y que las peticiones de memoria se realicen satisfactoriamente. Lo hemos conseguido con la ejecución simultánea de dos programas. Pero hemos encontrado una inconveniencia para la ejecución de más de dos aplicaciones. En tiempo de construcción de la máquina virtual, ella misma se utiliza en el proceso y dictamina de antemano los espacios de memoria de cada procesador virtual, por lo que dificulta la ejecución de un número indeterminado de programas, ya que no conocemos en el momento de creación del JikesRVM el número de aplicaciones que se ejecutarán con ella. Esto no ha sido un problema para nuestro proyecto, porque, ya se vaya a ejecutar una sola aplicación o dos, la máquina reservará los espacios de memoria para dos aplicaciones, aunque en uno de los casos la mitad de espacios no se utilizará.

A continuación mostraremos los comentarios que hemos hecho que mostrara la máquina virtual, utilizados para saber que el funcionamiento era correcto.

Una vez arrancado JikesRVM muestra todos los espacios de memoria que ha reservado antes de ejecutar ninguna aplicación. Los tres últimos espacios son los añadidos tras las modificaciones realizadas sobre el gestor de memoria.

```
VMResource 0 0x55000000 0x57000000 Meta data
VMResource 1 0x43000000 0x53000000 Boot
VMResource 2 0x53000000 0x55000000 Immortal
VMResource 3 0x74a00000 0x8b700000 Nursery
VMResource 4 0x5dd00000 0x74a00000 MS
VMResource 5 0x57000000 0x5dd00000 LOS
VMResource 6 0xa9100000 0xbfe00000 Nursery2
VMResource 7 0x92400000 0xa9100000 MS2
VMResource 8 0x8b700000 0x92400000 LOS2
```

Una vez que empieza a ejecutar los dos programas, mostramos por pantalla cada momento en el cuál se accede al gestor de memoria para alojar un objeto, visualizando la aplicación que hace la petición y la dirección de memoria en la que se guarda el objeto. Enseñamos cuatro líneas de salida.

```
allocateArray de programa 1 a la direccion 0x74bcfec0
```

```
allocateArray de programa 1 a la direccion 0x74bcfeec
allocateArray de programa 2 a la direccion 0xa910c6b0
allocateArray de programa 2 a la direccion 0xa910c730
```

Las dos primeras líneas se corresponden con una petición de la primera aplicación al gestor de memoria. Como podemos apreciar, la dirección que se ha utilizado pertenece a un espacio de memoria del primer programa. Las dos siguientes pertenecen a peticiones de la segunda aplicación. La dirección que se devuelve pertenece a un espacio del segundo programa. Esto verifica el buen resultado de las modificaciones añadidas al gestor de memoria.

5. Uso del JikesRVM multiaplicación

JikesRVM ejecuta las instrucciones "bytecode" (estas instrucciones son para la máquina virtual de Java) desde los archivos .class . El no compila el código de Java. Por eso se puede compilar todos los archivos fuentes a "bytecode" usando cualquier compilador de Java favorito.

A la hora de ejecutar una aplicación en JikesRVM, tendremos que usar la siguiente línea de comandos:

rvm [*rvm options...*] class [*args...*]

Sin embargo, si lo que queremos es ejecutar dos aplicaciones en JikesRVM, tendremos que usar la siguiente línea de comandos:

rvm [*rvm options...*] class1 [*args...*] @ class2 [*args...*]

Las opciones de ejecución de JikesRVM se podrán dividir en dos tipos: las estándar y no-estándar. Estas últimas tendrán que estar precedidas por "-x:". A continuación señalaremos las más importantes.

ESTÁNDAR

{ -cp | -classpath } : indicamos la dirección donde tiene que buscar las clases de la aplicación.

-D<name>=<value> : asignamos una propiedad del sistema.

-verbose:[class | gc | jni] : activamos el verbose.

-version : nos muestra la versión de la VM y terminamos la ejecución.

-showversion : nos muestra la versión de la VM y continuamos la ejecución.

-fullversion : es exactamente igual que "-version", pero nos muestra más información.

-? o -help : nos muestra un mensaje de ayuda.

-jar : se ejecuta un archivo jar.

NO-ESTÁNDAR

-X : nos muestra por pantalla información acerca de las opciones no-estándar.

-X:verbose : durante la ejecución nos muestra distinta información adicional.

-X:verboseBoot=<number> : nos muestra por pantalla información adicional mientras la VM está arrancando . Mediante el párametro <number>estableceremos el nivel de información.

-Xms<number>[<unit>] : indicamos el tamaño inicial del heap , siendo <number><units>de bytes.

-Xmx<number>[<unit>] : asignamos el tamaño máximo del heap , siendo <number><units>de bytes.

-X:sysLogfile=<filename> : redireccionamos los mensajes que puedan ir a "standard error" y los almacenamos en el archivo que metemos por parametro <file>.

-X:i=<filename> : leemos el arranque de JikesRVM desde <filename>.

-X:vm[:help] : nos muestra las opciones que tolera el núcleo de la VM.

-X:vm:<option> : nos transfiere la opción que metemos por parámetro, <option>, al núcleo de la maquina virtual.

-X:gc[:help] : nos muestra las opciones que tolera el administrador de memoria.

-X:gc:<option> : nos transfiere la opción que metemos por parámetro, <option>, al administrador de memoria.

-X:aos[:help] : nos muestra las opciones toleradas por el AOS (adaptive optimization system).

-X:aos:<option> : nos transfiere <option>al AOS.

-X:irc[:help] : nos muestra las opciones admitidas por el IRC (initial runtime compiler).

-X:irc:<option> : nos transfiere <option>al IRC.

-X:recomp[:help] : nos muestra las opciones toleradas por los compiladores destinados a la recompilación.

-X:recomp:<option> : nos transfiere <option>a los compiladores destinados a la recompilación.

-X:base[:help] : nos muestra las opciones toleradas por el "baseline compiler"".

-X:base:<option> : nos transfiere <option>al "baseline compiler".

-X:quick[:help] : nos muestra las opciones toleradas por el "quick compiler".

-X:quick:<option> : nos transfiere <option>al “quick compiler”.

-X:opt[:help] : nos muestra las opciones toleradas por el “optimizing compiler””.

-X:opt:<option> : nos transfiere <option>al “optimizing compiler”.

-X:prof:<option> : nos transfiere <option>al ”profiling subsystem“.

-X:vmClasses=<path> : nos carga las clases desde <path>, siendo <path>el nombre de un archivo, de varios archivos o incluso de archivos .jar . Los nombres de archivos que indicamos en <path>tienen que estar separados por (“:”).

-X:cpuAffinity=<int> : determinamos mediante el parametro <int>la CPU (física), a la cual será vinculada el primer procesador virtual.

6. Benchmarking

6.1. Descripción de los benchmarks utilizados

En esta sección describiremos los benchmarks utilizados para hacer las pruebas.

_200_ check. Es un programa simple que prueba varias características de la máquina virtual Java para asegurarse que esta proporciona un entorno conveniente para programas escritos en Java. Incluye:

- Indexado con arrays.
- Crea una superclase y su subclase y entonces accede a varibles públicas, privadas y protegidas y a métodos sobrecargados.
- Chequea si la máquina virtual realiza desordenadamente suposiciones sobre métodos no finales. Hay una clase y una subclase, las cuáles implementan un método de distinta manera. Por tanto, si la máquina virtual usa la implementación de la superclase, se obtiene un resultado erróneo.
- Un conjunto de pruebas llamadas *PepTest* donde dichas pruebas son ejecutadas para probar los siguientes aspectos: ejecución de if-then-else, operaciones con arrays, bucles, divisiónes, etc.

_201_ compress. Modificación del método Lempel-Ziv (LZW). Básicamente encuentra subcadenas comunes y las reemplaza con un código de tamaño varible. Este es determinista, y puede realizarse al vuelo. Pero, el proceso de descompresión no necesita ninguna tabla de entrada, pero recupera la tabla que fue construida.

_202_ jess. JESS es un un sistema experto basado en el sistema experto CLIPS de la NASA. De forma sencilla, constituye en aplicar declaraciones if-then, llamadas reglas, a un conjunto de datos, llamados lista de hechos. Este berchmark realiza el trabajo de resolver una serie de puzzles comunes con CLIPS. Para incrementar el tiempo de ejecución del problema, interactivamente aserta un nuevo conjunto de hechos que representan el mismo puzzle pero con diferentes literales. Los hechos antiguos no son eliminados. Pero el motor de inferencia debe buscar progresivamente a lo largo de las reglas y proceder con su ejecución.

_205_ raytrace. Es un raytracer que trabaja sobre una escena que representa un dinosaurio.

_209_ db. Ejecuta un motón de funciones de bases de datos sobre una base de datos en memoria, lee de un archivo de 1MB, el cuál contiene registros

con nombres, direcciones y números de teléfono de individuos, y un archivo de 19KB llamado scr6, el cuál contiene un conjunto de operaciones a realizar sobre los registros en el archivo.

_227_ mtrt. Es una variante del *_205_ raytrace*, un "raytracer" que trabaja sobre una escena en la que se representa un dinosaurio, donde dos hilos renderizan la misma escena.

_228_ jack. Un generador de parsers para Java basado en el conjunto de herramientas para construcción de compiladores Purpude (PCCTS). Esta es una versión temprana del que se llama ahora JavaCC. La carga de trabajo consiste de un archivo llamado jack.jack, el cuál contiene instrucciones para la generación del mismo jack.

6.2. Tablas de tiempos y gráficas

En esta sección se adjuntan y comentan las tablas de tiempos conseguidas a través de los benchmarks. Además se incorpora una gráficas para ver mejor los resultados. En las pruebas se han medido varios tiempo.

Tiempo de usuario. Tiempo en la máquina virtual se está ejecutando en modo usuario.

Tiempo de sistema. Tiempo en el que máquina virtual se está ejecutando en modo sistema, por ejemplo, tras una llamada al sistema operativo.

Tiempo real. Tiempo real de ejecución de la máquina virtual.

Las tres primeras tablas muestran los tiempos de ejecución de cada benchmark por la máquina virtual original. Primero se muestra la tabla de los tiempos reales.

Jess	Raytrace	Mtrt	Jack
0m3.626s	0m11.543s	0m9.855s	0m12.857s

Ahora mostramos los tiempos de usuario.

Jess	Raytrace	Mtrt	Jack
0m2.456s	0m11.260s	0m9.456s	0m11.445s

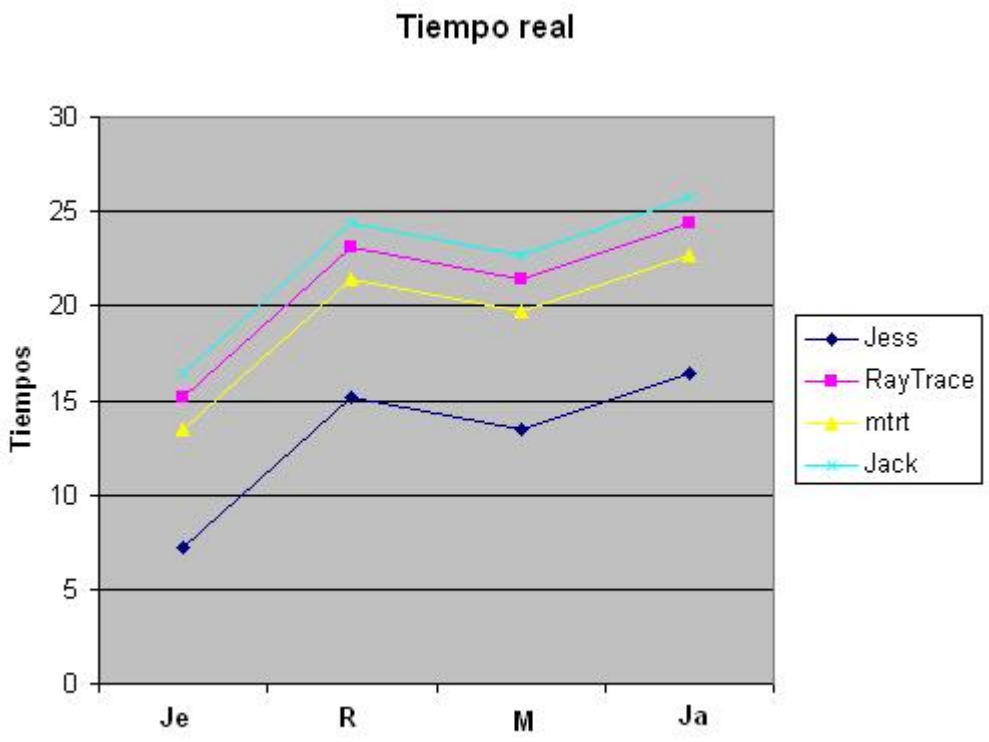
Por último los tiempos de sistema.

Jess	Raytrace	Mtrt	Jack
0m1.173s	0m0.300s	0m0.246s	0m1.422s

A través de las anteriores tablas y los tiempos calculados, mostramos las tablas correspondientes a la ejecución de dos máquinas virtuales, cada una de ellas ejecutando un benchmark. Los datos mostrados se corresponden a la suma del tiempo de una máquina virtual en ejecutar el primer benchmark y el tiempo de otra máquina virtual ejecutando el segundo benchmark, utilizando los datos anteriores. Acompañamos cada tabla con su gráfica correspondiente.

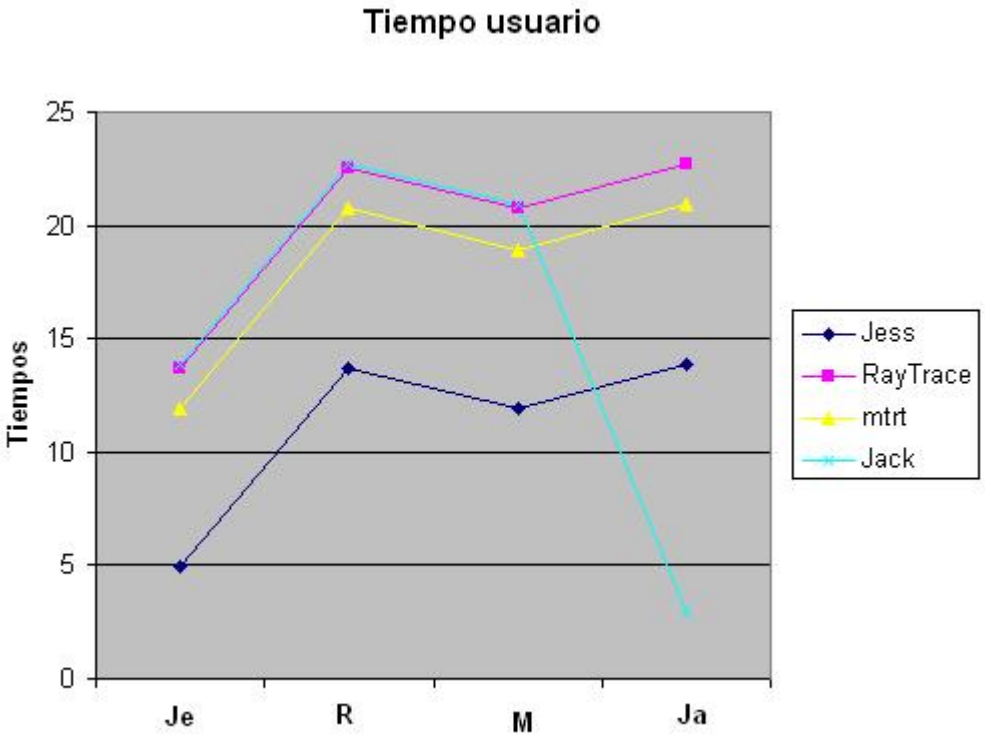
Tiempos reales.

	jess	raytrace	mtrt	jack
jess	0m7.252s	0m15.169s	0m13.481s	0m16.483s
raytrace	0m15.169s	0m23.086s	0m21.398s	0m24.400s
mtrt	0m13.481s	0m21.398s	0m19.710s	0m22.712s
jack	0m16.483s	0m24.400s	0m22.712s	0m25.714s



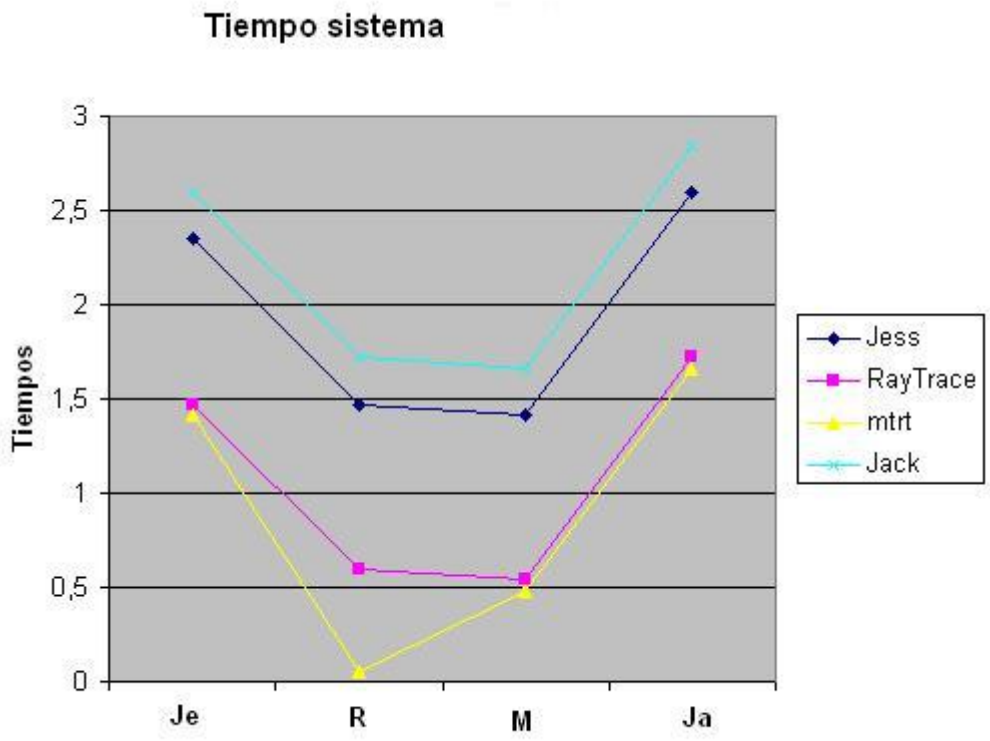
Tiempos de usuario.

	Jess	Raytrace	Mtrt	Jack
Jess	0m4.912s	0m13.716s	0m11.912s	0m13.901s
Raytrace	0m13.716s	0m22.520s	0m20.716s	0m22.705s
Mtrt	0m11.912s	0m20.716s	0m18.912s	0m20.901s
Jack	0m13.901s	0m22.705s	0m20.901s	0m2.890s



Tiempos de sistema.

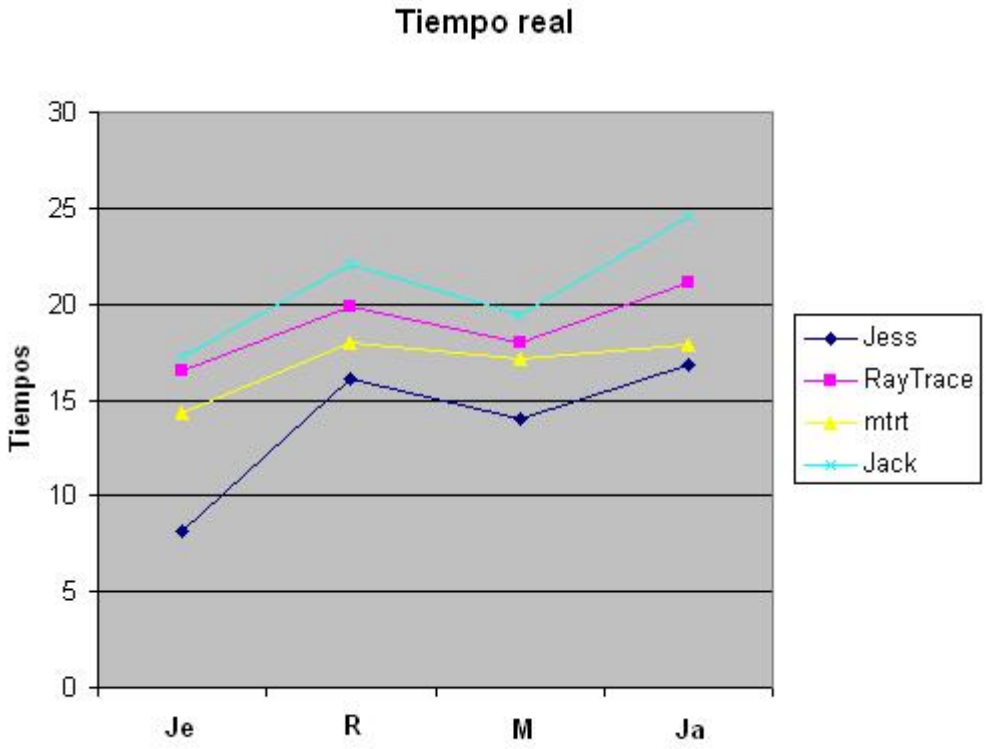
	Jess	Raytrace	Mtrt	Jack
Jess	0m2.346s	0m1.473s	0m1.415s	0m2.595s
Raytrace	0m1.473s	0m0.600s	0m0.542s	0m1.722s
Mtrt	0m1.415s	0m0.542s	0m0.484s	0m1.664s
Jack	0m2.595s	0m1.722s	0m1.664s	0m2.844s



A continuación mostramos las tablas conseguidas al ejecutar dos benchmarks sobre la máquina virtual multiaplicación implementada.

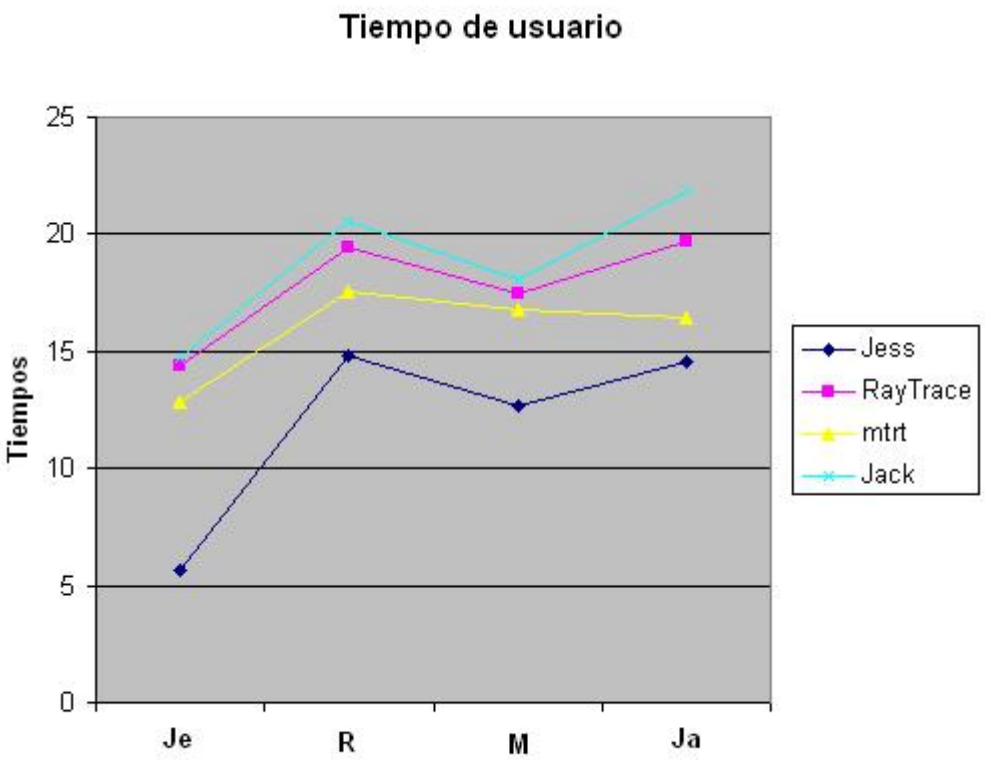
Tiempos reales.

	Jess	Raytrace	Mtrt	Jack
Jess	0m8.155s	0m16.118s	0m14.006s	0m16.780s
Raytrace	0m16.947s	0m19.879s	0m17.953s	0m21.090s
Mtrt	0m14.309s	0m17.994s	0m17.187s	0m17.835s
Jack	0m17.230s	0m22.085s	0m19.426s	0m24.559s



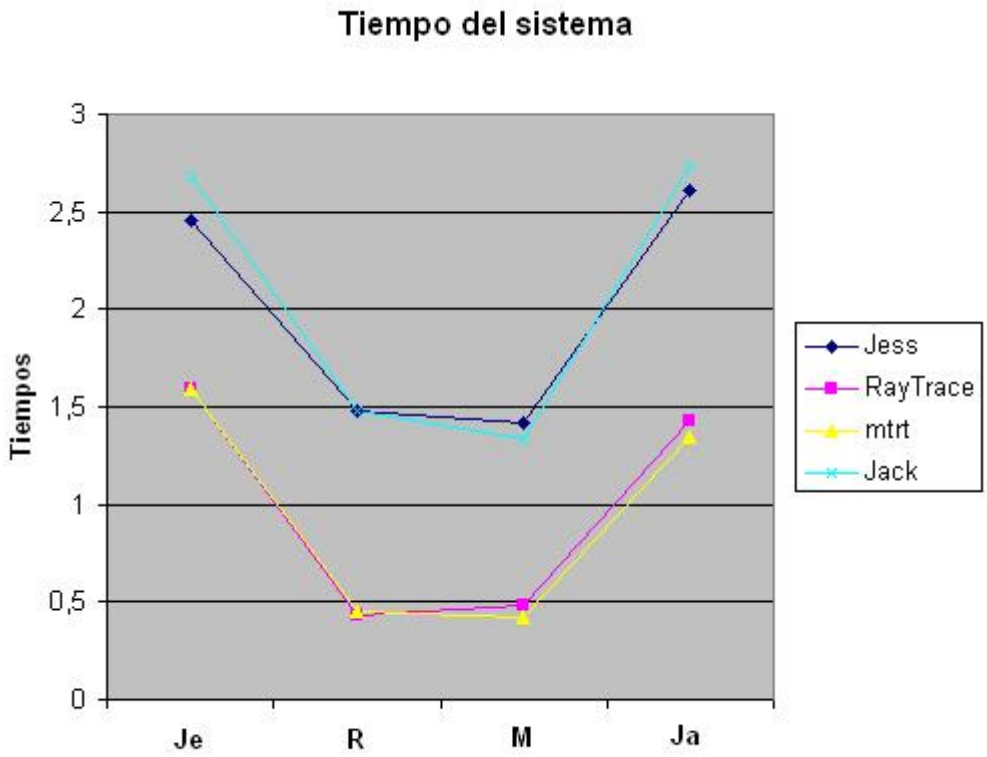
Tiempos de usuario.

	Jess	Raytrace	Mtrt	Jack
Jess	0m5.677s	0m14.831s	0m12.706s	0m14.521s
Raytrace	0m14.420s	0m19.471s	0m17.472s	0m19.673s
Mtrt	0m12.821s	0m17.573s	0m16.780s	0m16.467s
Jack	0m14.759s	0m20.574s	0m18.024s	0m21.819s



Tiempos de sistema.

	Jess	Raytrace	Mtrt	Jack
Jess	0m2.456s	0m1.582s	0m2.706s	0m2.668s
Raytrace	0m1.596s	0m0.429s	0m0.487s	0m1.431s
Mtrt	0m1.591s	0m0.447s	0m0.419s	0m1.346s
Jack	0m2.686s	0m1.481s	0m1.331s	0m2.736s



6.3. Consideraciones

Tras la ejecución de las pruebas y la representación de los datos, haremos unos comentarios con respecto al rendimiento de la nueva máquina virtual multiaplicación.

La mejora con respecto a la original se da en la ejecución simultánea de aquellas aplicaciones que necesitan procesar y crear muchos datos, es decir, hacen poco uso de la memoria. Este es el caso, como vemos en las pruebas, del benchmark llamado raytracer (renderizar una imagen no hace mucho uso de la memoria). Se aprecia mejora cuando ejecutamos el mismo benchmark dos veces con la nueva máquina virtual con respecto a ejecutar dos veces con la máquina original.

La caída de rendimiento reside en los programas que sí hacen mucho uso de la memoria. Es debido a nuestra política de, al tener que duplicar los espacios, reducirlos a la mitad del tamaño original. En este caso, estos programas llenan más a menudo el espacio, con lo que conlleva que activan más a menudo el hilo recolector. Esto es una perdida de eficiencia y tiempo de ejecución. Podemos apreciarlo con el caso del benchmark Jess. Este sistema

experto crea muchos hechos a partir de las reglas que intenta aplicar. Nuestra máquina, al ejecutar dos aplicaciones de este benchmark, empeoran en tiempo de ejecución con respecto al supuesto de que dos máquina originales ejecuten estas aplicaciones.

7. Ejemplos de código

7.1. VM_Thread.java

La parte principal de código modificado sobre el sistema de hilos se encuentra en la clase *VM_Thread*. A continuación mostraremos las nuevas constructoras de dicha clase que permiten asociar a un hilo el identeficador de su aplicación.

```
public class VM_Thread implements VM_Constants,
   VM_Uninterruptible {

  /**
  * Id del programa al que pertenece el hilo.
  */
  protected int idprograma;

  .
  .
  .

  public VM_Thread (int idprog) {
     this(MM_Interface.newStack(STACK_SIZE_NORMAL, false), idprog);
  }

  .
  .
  .

  public VM_Thread (byte[] stack, int idprog) {
     this.stack = stack;
     this.idprograma = idprog;

     VM.sysWrite(``Hilo creado para el programa '', idprograma
        + ``\n'');

     chosenProcessorId = (VM.runningVM ?
        VM_Processor.getCurrentProcessorId() : 0);
     suspendLock = new VM_ProcessorLock();
```

```
    contextRegisters            = new VM_Registers();

    .
    .
    .

  }
```

7.2. Plan.java

La parte principar de código modificada sobre el gestor de memoria, en el caso de nuestro proyecto el gestor CopyMS, se encuentra en la clase *Plan.java*. A continuación mostraremos la duplicación de recursos y algunas funciones importantes dentro del gestor.

```
public class Plan extends StopTheWorldGC implements
   VM_Uninterruptible {

   .
   .
   .

   // virtual memory resources
   private static MonotoneVMResource nurseryVM;
   private static FreeListVMResource msVM;
   private static FreeListVMResource losVM;

   //virtual memory resources para la otra aplicaci
   private static MonotoneVMResource nurseryVM2;
   private static FreeListVMResource msVM2;
   private static FreeListVMResource losVM2;

   // memory resources
   private static MemoryResource nurseryMR;
   private static MemoryResource msMR;
   private static MemoryResource losMR;

   // memory resources para otra aplicaci
   private static MemoryResource nurseryMR2;
```

```
private static MemoryResource msMR2;
private static MemoryResource losMR2;

// Mark-sweep collector (mark-sweep space, large objects)
private static MarkSweepSpace msSpace;
private static TreadmillSpace losSpace;
private static MarkSweepSpace msSpace2;
private static TreadmillSpace losSpace2;

.
.
.

public final VM_Address alloc(int bytes, boolean isScalar,
   int allocator, AllocAdvice advice) throws VM_PragmaInline {

  if (VM_Interface.VerifyAssertions) VM_Interface._assert(bytes ==
     (bytes & (~(BYTES_IN_ADDRESS-1))));
  VM_Address region;

  if(VM_Thread.getCurrentThread().getIdProg() == 1) {
     if (allocator == NURSERY_SPACE && bytes > LOS_SIZE_THRESHOLD) {
        region = los.alloc(isScalar, bytes);
     } else {
        switch (allocator) {
        case  NURSERY_SPACE: region = nursery.alloc(isScalar, bytes);
           break;
        case       MS_SPACE: region = ms.alloc(isScalar, bytes, false);
           break;
        case      LOS_SPACE: region = los.alloc(isScalar, bytes);
           break;
        case IMMORTAL_SPACE: region = immortal.alloc(isScalar, bytes);
           break;
        default:
           if (VM_Interface.VerifyAssertions)
              VM_Interface.sysFail(``No such allocator'');
           region = VM_Address.zero();
        }
     }
  } else {
     if (allocator == NURSERY_SPACE && bytes > LOS_SIZE_THRESHOLD) {
```

```
            region = los2.alloc(isScalar, bytes);
         } else {
            switch (allocator) {
            case  NURSERY_SPACE: region = nursery2.alloc(isScalar, bytes);
               break;
            case       MS_SPACE: region = ms2.alloc(isScalar, bytes, false);
               break;
            case      LOS_SPACE: region = los2.alloc(isScalar, bytes);
               break;
            case IMMORTAL_SPACE: region = immortal.alloc(isScalar, bytes);
               break;
            default:
               if (VM_Interface.VerifyAssertions)
                  VM_Interface.sysFail(``No such allocator'');
               region = VM_Address.zero();
               }
        }
     }
     if (VM_Interface.VerifyAssertions) Memory.assertIsZeroed(region, bytes);
        return region;
  }

  .
  .
  .

  public static final VM_Address traceObject(VM_Address obj) {
     if (obj.isZero()) return obj;
     VM_Address addr = VM_Interface.refToAddress(obj);
     VM_Address aux;
     byte space = VMResource.getSpace(addr);
     if(VM_Thread.getCurrentThread().getIdProg()==1){
        switch (space) {
        case NURSERY_SPACE:  return CopySpace.traceObject(obj);
                 case MS_SPACE:
           try {
              aux =  msSpace.traceObject(obj, VMResource.getTag(addr));
           } catch (Exception e) {
               aux = msSpace2.traceObject(obj, VMResource.getTag(addr));
           }
           return aux;
```

```
        case LOS_SPACE:
           try {
              aux = losSpace.traceObject(obj);
           } catch (Exception e) {
              aux = losSpace2.traceObject(obj);
           }
           return aux;
        case IMMORTAL_SPACE: return ImmortalSpace.traceObject(obj);
        case BOOT_SPACE:     return ImmortalSpace.traceObject(obj);
        case META_SPACE:     return obj;
        default:
           if (VM_Interface.VerifyAssertions)
              spaceFailure(obj, space, ``Plan.traceObject()'');
           return obj;
        }
     }
     else{
        switch (space) {
           case NURSERY_SPACE:  return CopySpace.traceObject(obj);
           case MS_SPACE:       return msSpace2.traceObject(obj,
              VMResource.getTag(addr));
           case LOS_SPACE:      return losSpace2.traceObject(obj);
           case IMMORTAL_SPACE: return ImmortalSpace.traceObject(obj);
           case BOOT_SPACE:     return ImmortalSpace.traceObject(obj);
           case META_SPACE:     return obj;
           default:
              if (VM_Interface.VerifyAssertions)
                 spaceFailure(obj, space, ``Plan.traceObject()'');
              return obj;
          }
     }
  }

  .
  .
  .

}
```

8. Conclusiones

A lo largo del desarrollo de este proyecto hemos comprendido poco a poco el funcionamiento de una máquina virtual, esto es debido a que gran parte del proyecto ha consistido en investigar los entresijos de JikesRVM.

La máquina se compone de una cantidad enorme de clases de una complejidad considerable, lo que ha hecho alargar casi hasta los últimos meses la fase de investigación. La investigación ha tenido 4 fases, la primera se basó en comprender el funcionamiento general, la segunda el comportamiento en concreto de los hilos, despés de la memoria y por último concretar cuales eran los recursos útiles para nuestro objetivo.

Simultáneamente a la investigación se pudo hacer avances en la implementación, pero la mayoría de ellos fueron realizados en los últimos meses. La cantidad de código escrito es más bien pequeña, comparada con la que se ha escrito en otros proyectos, esto es porque la verdadera complicación era saber que código escribir y donde, ya que había que saber cual era la clase apropiada para recibir los cambios además esta debía ocasionar el menor número de consecuencias en el resto de las clases.

Este proyecto es fácilmente ampliable, se podría hacer la máquina multiaplicación pero con n aplicaciones, nosotros nos planteamos que la igual que hacíamos para dos sería fácil hacerlo para n, el problema nos lo encontramos en que la máquina reserva el espacio de memoria antes de saber cuantas aplicaciones tiene que ejecutar, la posible ampliación consistiría en intentar posponer esa reserva de memoria hasta conocer el número de aplicaciones a ejecutar.

Finalmente podemos decir que hemos conseguido nuestro objetivo, convertir la máquina virtual Jikes en multiaplicación de una forma eficiente. Ya que disminuye tanto en tiempo de ejecución como en memoria respecto a la máquina original.

9. Palabras claves

Gestor de memoria

Hilo

JikesRVM

Máquina virtual de Java

Memory manager

Multiaplicación

Thread

10. Bibliografía

Referencias

[1] Javadoc de JikesRVM

[2] http://jikesrvm.sourceforge.net/userguide/HTML/userguide_0.html

[3] "Processes in KaffeOS:Isolation, Resource Management, and Sharing in Java" Godmar Back, Wilson C. Hsieh, Jay Lepreau

[4] http://www.artima.com/insidejvm/ed2/jvmP.html

[5] www.ibm.com/developerworks/java/library/j-jalapeno/

11. Autorización

Mediante el documento aquí firmado autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código del proyecto "Máquina virtual de Java multiaplicación"

Fdo:	**Fdo:**	**Fdo:**
Jose Carlos Alján DNI: 50882863	Ricardo González DNI: 53438842	Rebeca Shakar DNI: 53447850