La máquina virtual de Java: Sincronización de hilos.

Por: Barrios López, Francisco
Universidad Nacional Autónoma de México
Facultad de Ingeniería
Abril, 2022

RESUMEN– Muchos de nosotros hemos usado el lenguaje de programación Java, y muchos sabemos que todos los programas Java se ejecutan dentro de una computadora abstracta conocida como la Máquina Virtual Java. En realidad, la MVJ corre mucho más que solo el lenguaje de programación Java, y lo hace en conjunto con otras tecnologías.
Pues bien, uno de los fuertes del lenguaje Java es la posibilidad del *multithreading* a nivel del lenguaje mismo. Esto es posible gracias a la sincronización: se lleva a cabo la coordinación de procesos y se controla el acceso a datos entre múltiples procesos.
En este escrito se intentará ver como funciona la sincronización de hilos en la MVJ, así como una breve introducción sobre los *Opcodes* propios de la MVJ que hacen posible esto (desde el código compilado a ensamblador de la MVJ), acompañada de un pequeñísimo panorama sobre la arquitectura Java.

1. Introducción.

Antes de comenzar, se necesita saber ¿En dónde estoy? O más bien, ¿Dónde se ubica la MVJ dentro de la arquitectura de Java? La arquitectura de Java cuenta con cuatro tecnologías distintas, relacionadas entre sí:

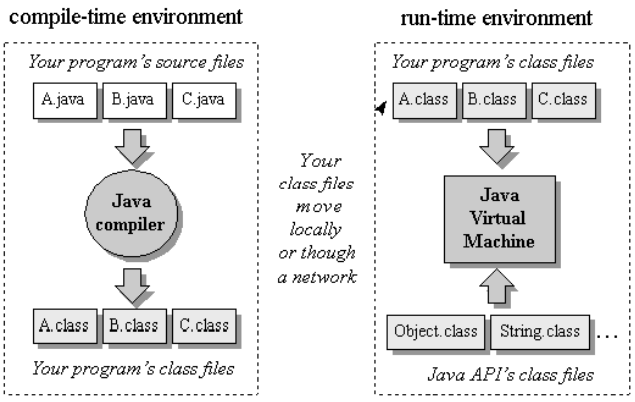

Imagen 1: Entorno de programación de Java [Imagen] por Bill Venners, 2019, artima (https://www.artima.com/insidejvm/ed2/images/fig1-1.gif)

Un programa Java esta expresado en archivos escritos en *el lenguaje de programación Java*, para posteriormente ser compilados dando como resultado a *los archivos de clase Java*, y ejecutados por *la Máquina Virtual de Java*. Antes de todo esto, el programa debe de ser escrito, en donde se obtiene el acceso a determinados recursos del sistema, llamando a métodos de clases que utilizan *la interfaz de programación de aplicaciones Java* (API Java).

El conjunto entre la Maquina Virtual Java y la API de Java forman la *Java Platform*. Los programas Java pueden ser ejecutados en cualquier tipo de computadora, ya que la plataforma de Java puede ser implementada en Software.

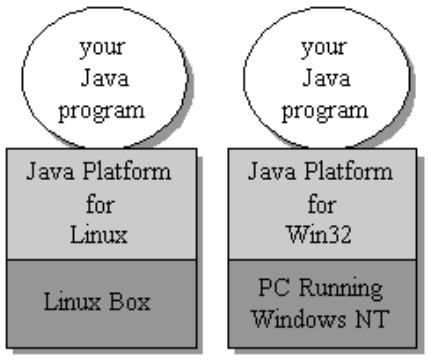

Imagen 2: Plataformas Java [Imagen] por Bill Venners, 2019, artima (https://www.artima.com/insidejvm/ed2/images/fig1-2.gif)

2. Monitores.

Para admitir la sincronización, Java utiliza al monitor. Para el monitor de Java, existen dos tipos de sincronización: *exclusión mutua* y *cooperación*.

La *exclusión mutua* es implementada mediante el bloqueo de objetos, y se refiere a que varios subprocesos pueden estar trabajando independientemente con datos compartidos, sin ninguna interacción entre ellos. La *cooperación*, en cambio, permite que los subprocesos puedan trabajar juntos para lograr un objetivo en común.

En Java, existen muchas formas de implementar métodos protegidos. Cada objeto entonces tiene un bloqueo, así como un conjunto de espera, que solo es posible manipularlo con los métodos *wait()*, *notify()*, *notifyAll()* y *Thread.interrupt()*. Entonces podemos definir al *monitor* como aquellas entidades que poseen bloqueos y conjuntos de espera, y por tanto, cualquier objeto puede servir como *monitor*. La MVJ guarda para cada objeto su propio conjunto de espera. Cada conjunto puede contener subprocesos bloqueados, esperando a que se invoque una *notificación* o a que se *libere* la espera.

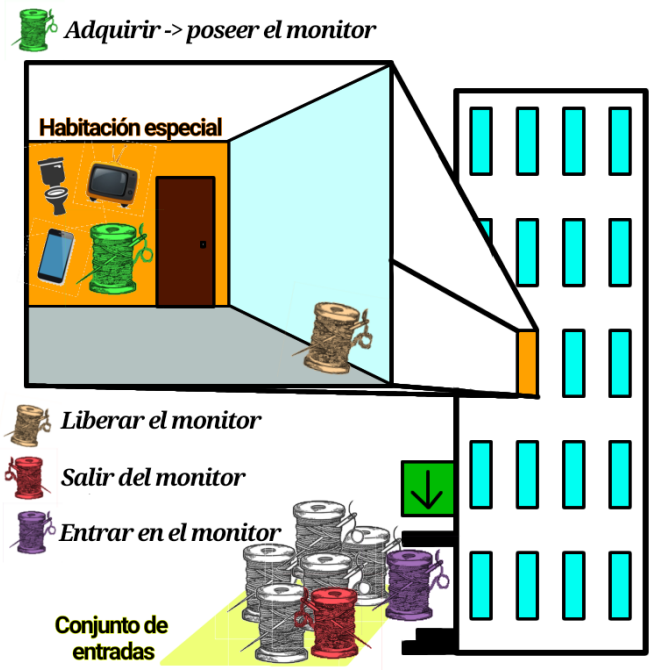

Imagen 3 Una forma meramente ilustrativa de ver al monitor

El monitor esta asociado con bits de datos, y con bits de código (*regiones del monitor*, *monitor regions*). Una región del monitor es un código que debe ejecutarse como indivisible con respecto a un monitor. Para que un subproceso pueda entrar al monitor, debe de llegar al principio de la *región del monitor*, y para que pueda ejecutarlo, el subproceso debe adquirir el monitor. El subproceso tiene acceso exclusivo a los datos hasta que se libere el monitor.

Cuando un proceso nuevo llega al monitor, y no hay otro hilo que tenga adquirido el monitor, o esperando en el *conjunto de entradas*, entonces el proceso puede continuar. Por el contrario, si llega, y hay más hilos esperando en el *conjunto de entradas*, entonces debe de esperar hasta que se libere el monitor. Cuando esto pasa, el hilo debe competir con el resto de los hilos que estén esperando, y el hilo ganador será el que pueda adquirir el monitor.

2.1 Monitor de la MVJ

La MVJ utiliza un tipo de monitor conocido como *Wait and notify*. En este tipo, un subproceso que posee el monitor puede suspenderse por medio de un comando *wait*. Cuando el subproceso ejecuta una espera, libera al monitor y entra en un *conjunto de espera*. El hilo esperará hasta que otro hilo, realice una notificación (*notify*) dentro del monitor. El hilo notificante seguirá con el monitor adquirido, y saldrá de él por su cuenta, ya sea ejecutando una *espera* o terminando su proceso. Una vez que el hilo notificante libere el monitor, el hilo en espera *resucitará* y volverá a adquirir el monitor.

Ahora bien, si el subproceso que tiene adquirido al monitor no lanza una notificación a los hilos en espera antes de liberar el monitor, y ninguno de los hilos en espera fue notificado, entonces solo los hilos en el *conjunto de entradas* competirán para adquirir el monitor. Si el hilo propietario si lanzó la notificación, entonces los hilos que están en el *conjunto de entradas* competirán con uno o más hilos del *conjunto de espera* para adquirir el monitor.

Una posible representación del monitor Java es el siguiente:

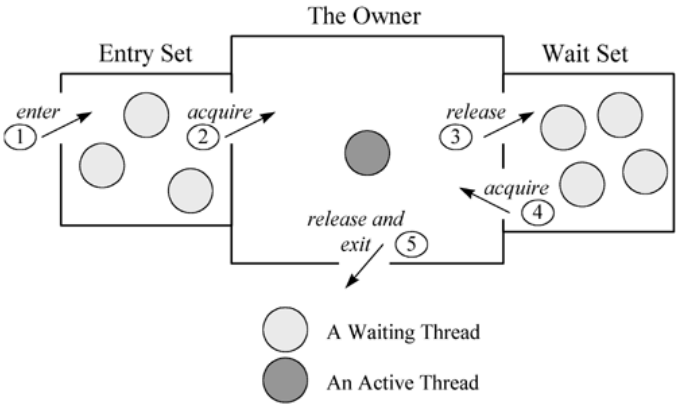

Imagen 4 Un monitor Java [Imagen] por Bill Venners, 2019, artima (https://www.artima.com/insidejvm/ed2/images/fig20-1.gif)

En la MVJ, los hilos que vayan al *conjunto de espera* pueden especificar un tiempo de espera. Si el hilo en espera no recibe una notificación en el tiempo especificado, la MVJ le lanza la notificación automáticamente, y el hilo resucitará. La MVJ ofrece dos comandos de notificación: *notify* selecciona

aleatoriamente un hilo del *conjunto de espera* y lo marca para resucitarlo; y *notify all* marca a todos los hilos en espera para su eventual resucitación.

La forma en que la MVJ selecciona el siguiente subproceso a adquirir el monitor depende de las implementaciones que le quiera dar el programador. Podría usarse una implementación FIFO (o *First Come, First Serve*); o varias colas FIFO, una por cada nivel de prioridad; o una pila LIFO (o *Last Come, First Serve*). En fin, al gusto de cada quién.

3. Object Locking

Un diagrama de bloques de la MVJ:

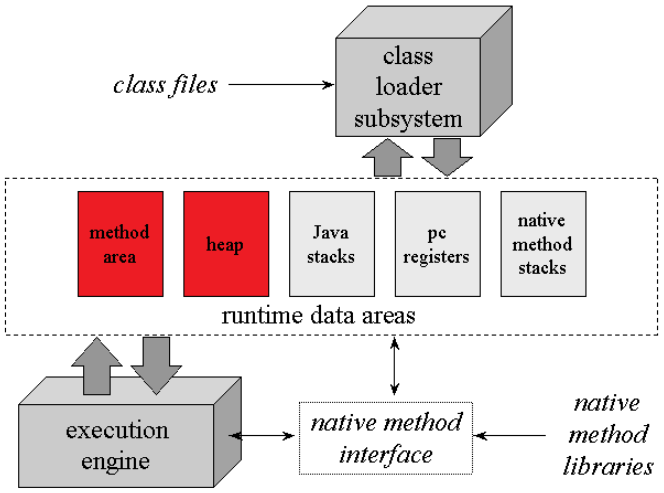

Imagen 5 Arquitectura de la MVJ [Imagen] por Bill Venners, 2019, artima (https://www.artima.com/insidejvm/ed2/images/fig5-1.gif)

En la MVJ, el *heap* y el *method area* son compartidos por todos los subprocesos, por lo que los programas Java deben de coordinar el acceso a las variables de instancia (almacenadas en el *heap*) y a las variables de clase (almacenadas en el *method area*).

Cada objeto y clase viene asociado lógicamente con un monitor. Para implementar la exclusión mutua, la MVJ asocia un bloqueo (mutex). Bloquear un objeto significa *adquirir el monitor* de ese objeto.

Un subproceso puede bloquear el mismo objeto varias veces, y solo el subproceso que haya adquirido al monitor puede bloquear el objeto varias veces. Para cada objeto, la MVJ cuenta cuantas veces ha sido bloqueado un objeto. El conteo comienza en 0, y cada vez que el objeto se bloquea, el número se incrementa de uno en uno. Cada vez que el bloqueo se libera, el conteo va decrementándose, y hasta que el conteo llegue a 0, el monitor se libera y ya puede ser utilizado por otro subproceso.

Un subproceso solicita un bloqueo cuando este llega a la *región del monitor*. Cuando se llega a la primera instrucción, el subproceso debe de adquirir el monitor del objeto al que se hace referencia, y no podrá continuar con la ejecución del código hasta que adquiera el bloqueo. Una vez adquirido, el hilo entra al código protegido, y al salir del bloque, el bloqueo es liberado.

Como programador, no tenemos que preocuparnos sobre bloquear explícitamente un objeto, ya que esto lo implementa la MVJ. Lo que si debería de preocuparnos, es identificar las regiones *criticas* del código, y protegerlas escribiendo *instrucciones* y *métodos sincronizados*.

4. Ejemplo simple de compilación para instrucciones sincronizadas y su secuencia de bytecodes

La sincronización es implementada mediante la entrada y salida al monitor. El JDK de Oracle contiene un compilador el cual nos proporciona las instrucciones propias de la MVJ, y entre ellas encontramos a *monitorenter* (adquiere el bloqueo asociado al objeto) y *monitorexit* (libera el bloqueo asociado al objeto).

Hay que tomar en cuenta que toda instrucción de la MVJ generado por el JDK de Oracle toma la siguiente forma:

```
<índice> <opcode> [ <operando1> [
<operando2>... ]] [<comentario>]
```

El <índice> es quién contiene los bytes de código de la MVJ para un método. Se trata del mnemotécnico para el Opcode de la instrucción, y esta instrucción puede tener 0 o más operandos. Opcionalmente se tiene un comentario.

La MVJ es una MV orientada a la pila, y la mayoría de las operaciones toman operandos de la pilas de operandos, y/o envían resultados a la pila de operandos, y se hace de acuerdo a la trama actual. Cada nueva trama es creada cuando se invoca un método, y se crea una nueva pila de operandos.

El conjunto de instrucciones de la MVJ utiliza bytecodes distintos para las operaciones, y con esto

se distinguen los tipos de operandos. Muchos operandos son muy probables de utilizarse (como las constantes *-1, 0, 1, 2, 3, 4, 5*), por lo que estos operandos están implícitos en el operando, logrando que el código compilado sea más compacto y eficiente.

Ahora si con el ejemplo para declarar un bloque de instrucciones sincronizadas, si tenemos como ejemplo el siguiente método:

```
void metodo (Foo f){
      synchronized (f) {
            HazAlgo();
      }
}
```

Tenemos la compilación para la MVJ de la siguiente forma (del lado izquierdo se tienen los *Opcode* y *mnemotécnicos*, y del lado derecho se tienen comentarios) [versión JDK 1.0.2] :

```
Method void metodo(Foo)
0   aload_1                    // Empuja a f
1   dup                  // Se duplica en la
                                        pila
2   astore_2               // Se guarda el
                           duplicado en la
                          variable local 2
3   monitorenter             // Se entra al
                          monitor asociado
                           con el objeto f
4   aload_0               // Con el monitor
                    adquirido, se sigue a
                la siguiente instrucción
5   invokevirtual #5           // Se llama a
                                 HazAlgo ();
8   aload_2                 // Se empuja la
                          variable local 2
9   monitorexit             // Se libera el
                          monitor asociado
                               al objeto f
10  goto 18               // Se complete el
                   método normalmente, y
                  se salta a la línea 18
13  astore_3            // En caso de error,
                              terminar aquí
14  aload_2                    // Empujar la
                          variable local 2
15  monitorexit          // Con error o no,
                        se debe de liberar
                                el monitor
16  aload_3              // Empujar el valor
                                   arrojado
17  athrow                // ...y volver a
                             lanzar el valor
                                al invocador
18  return              // Termina el método
                          en el caso normal,
                                  o erróneo
```

Del ejemplo observamos que la MVJ se asegura que, en cualquier forma de finalización del método (sea normal o abruptamente), se debe liberar el monitor con *monitorexit* para cualquier anterior posesión del monitor con *monitorenter*. Para lograr emparejar ambas instrucciones, el compilador debe generar controladores de excepciones, que al presentarse una excepción, su código asociado ejecutará las instrucciones *monitorexit* necesarias.

5. Métodos sincronizados (Sin ejemplo)

Si lo que se desea es sincronizar un método completo, se usa la siguiente sintaxis:

```
class nombre_clase{
    synchronized void nombre_método() {
        [Instrucciones]
    }
}
```

Para los métodos sincronizados, la MVJ no utiliza Opcodes especiales para invocar o devolver desde estos métodos. Cuando se resuelve la referencia simbólica de un objeto, se determina si el método está sincronizado. Si es así, la MVJ adquiere un *mutex* antes de invocar al método.

6. Referencias y bibliografía:

Bill Venners (2019) The Java Programming Environment [Imagen 1], artima. https://www.artima.com/insidejvm/ed2/images/fig1-1.gif

Bill Venners (2019) Java programs run on top of the Java Platform [Imagen 2], artima. https://www.artima.com/insidejvm/ed2/images/fig1-2.gif

Bill Venners (2019) A Java Monitor [Imagen 4], artima. https://www.artima.com/insidejvm/ed2/images/fig20-1.gif

Bill Venners (2019) The Internal Architecture of the Java Virtual Machine [Imagen 5], artima. https://www.artima.com/insidejvm/ed2/images/fig5-1.gif

Bill Venners (1998) *Thread Synchronization*. En *Inside de Java 2 Virtual Machine*. Segunda Edición. Computing McGraw – Hill. Disponible en artima en: https://www.artima.com/insidejvm/ed2/threadsynch.html

Bill Venners (1998) *Introduction to Java´s Architecture*. En *Inside de Java 2 Virtual Machine*. Segunda Edición. Computing McGraw – Hill. Disponible en artima en: https://www.artima.com/insidejvm/ed2/introarch2.html

Tim Lindholm (2022) *Compiling for the Java Virtual Machine*. En *The Java® Virtual Machine Specification. Java SE 18 Edition*. Disponible en docs Oracle en: https://docs.oracle.com/javase/specs/jvms/se18/html/jvms-3.html

Tim Lindholm (2022) *The Structure of the Java Virtual Machine*. En *The Java® Virtual Machine Specification. Java SE 18 Edition*. Disponible en docs Oracle en: https://docs.oracle.com/javase/specs/jvms/se12/html/jvms-2.html#jvms-2.10

Doug Lea (2000) *Monitor Mechanics*. (p 184). En *Concurrent Programming in Java ™. Design Principles and Patterns*. Segunda edición. Adisson Wesley Professional. 411pp. Previsualización en: https://books.google.com.mx/books?id=-x1S4neCSOYC&printsec=frontcover#v=onepage&q&f=false