Kipu: hilos ligeros para Java*

Michael Dorin**

Resumen

*En este artículo se describe un mecanismo cooperativo de planificación de hilos que puede usarse de manera efectiva en máquinas virtuales Java en tiempo real. Este planificador se implementa utilizando una cola de prioridades (*priority queue*) basada en montículo (*heap*) que permite un tiempo de procesamiento O(log N). Dado que este planificador es estrictamente cooperativo, la coordinación de los hilos se realiza sin el requisito de una sincronización formal de hilos Java. Aunque se pueden lograr resultados similares a través del diseño de programas, esta abstracción permite a los programadores enfocarse en el desarrollo de su aplicación.*

*Palabras clave: concurrencia, montículo (*heap*), Java, hilo, tarea*

Kipu – Lightweight threads for Java

Abstract

This paper describes a cooperative thread scheduling mechanism, which can be used effectively on real-time java virtual machines. This scheduler is implemented using a heap-based priority queue which allows for O(log N) processing time. Since this scheduler is strictly cooperative, thread coordination is performed without the requirement of formal java thread synchronization. Though similar outcomes can be achieved through program design, this abstraction allows developers to focus on their application.

Key words: concurrency, heap, Java, thread, task

* Traducción del inglés: Mercedes Vitteri Quiroz.

** Mi sincero agradecimiento a Scott McSpadden por su valiosa y generosa ayuda en la preparación de este proyecto.

Introducción

La introducción de *Android* ha revitalizado la programación embebida de tiempo real en Java. Mientras que Oracle tiene su propia JVM (Java Virtual Machine / Máquina Virtual de Java) diseñada para dispositivos embebidos, no siempre es una opción práctica. Además, varias JVM están disponibles para dispositivos muy pequeños. La mayoría de JVM embebidas no intentan soportar todas las características de Java y, en muchos casos características tales como el cambio de contexto, aun cuando se implementen, son de gran impacto para pequeños dispositivos.

Durante una clase, a principios del 2007, fue presentada una arquitectura de planificación cooperativa en la cual se podían crear y ejecutar las tareas según diferentes prioridades. Con la planificación cooperativa, los hilos se ejecutan hasta su término o voluntariamente liberan al procesador (CPU) a disposición de otros hilos.

La implementación en lenguaje "C" del material de diseño produjo un sistema operativo cooperativo extremadamente pequeño. Este sistema operativo necesita que todas las tareas se ejecuten hasta su conclusión, pero requiere de muy poco tiempo de procesamiento.

A medida que los dispositivos y herramientas se han vuelto más sofisticados, los programadores aparentemente entienden menos que antes la programación embebida en tiempo real. Los estudiantes pueden aprobar satisfactoriamente el curso de Sistemas Operativos sin llegar a comprender totalmente la noción de concurrencia, ni los problemas causados por ella. Cuando estos estudiantes empiezan su vida profesional crean aplicaciones construidas sobre "tapices de hilos", con poca idea de lo que puede afectar el movimiento de hilos al reinicio del sistema. Como profesor del curso de Estructuras de Datos me di cuenta de que podía demostrar la concurrencia a través de la combinación de estructuras de datos con un planificador cooperativo, y fue de esa manera cómo fue creado Kipu. Este puede ser útil en un entorno académico así como en dispositivos integrados Java de bajo consumo, por ejemplo NanoVM, que está diseñado para el procesador Atmel AVR ATmega, y no es compatible con hilos (http://sourceforge.net/projects/nanovm/ y http://www.harbaum.org/till/nanovm/ index.shtml) e incluso en dispositivos *Android*. Kipu se puede utilizar con la mayoría de las versiones de Java y se ha utilizado con éxito en Java 8.

1. Evaluación de los hilos en Java

El objetivo de este artículo no es cubrir los detalles internos sobre cómo funcionan exactamente los hilos de Java, ya que hay abundante información disponible sobre ello. Sin embargo, es importante mencionar que algo de información

sobre hilos en Java podría ser útil para comprender cómo Kipu puede beneficiar algunos entornos.

Un hilo es un "hilo de ejecución" de un programa y Java estándar permite que múltiples hilos se ejecuten de forma concurrente (https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html). La especificación de la máquina virtual Java define un modelo de hilos de ejecución que permite una amplia variedad de arquitecturas de hilos, incluyendo hilos nativos (Venners, 1999). Los hilos de Java tienen además su propia memoria de trabajo. Con esto en mente, uno debe recordar que el mecanismo de hilo aplicado quizá no sea el más eficiente para cada proyecto.

A cada hilo se le asigna una prioridad. Java puede ejecutar hilos con diez distintas prioridades y no hay garantía de segmentación de tiempo. Los hilos con más alta prioridad se ejecutan primero y los hilos con prioridades menores se ejecutan cuando los hilos con mayor prioridad completan el uso del procesador (Venners, 1999). Mucha gente no se da cuenta de esto y crea situaciones de interbloqueo, inversión de prioridades e inanición.

Java soporta la sincronización de hilos mediante bloqueo de objetos y el hilo espera y notifica (https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html). El bloqueo de objetos es análogo a la función "ingresar a la región crítica" que muchos Real-Time Operating Systems (RTOS) soportan y protegen la memoria, objetos, entre otros, lo cual no debe ser modificado por más de un hilo a la vez. El mecanismo de espera y notificación en un hilo permite las relaciones productor-consumidor entre los hilos.

Mientras que los hilos de Java son útiles, demasiados hilos pueden ser un problema. Después de un tiempo el programador empieza a agregar sincronización adicional para arreglar los problemas que va encontrando. El programa resultante es a menudo una "planificación cooperativa de hecho", sin haber sido esta la intención del programador.

2. Planificación cooperativa

Cuando un sistema operativo multitarea decide ejecutar una tarea/hilo diferente, guarda el contexto actual del hilo (es decir los registros) y carga el contexto del hilo que debe ejecutarse. Este proceso se denomina cambio de contexto y aumenta el tiempo de procesamiento de la aplicación en su conjunto (Labrosse, 1992). El tiempo que se requiere para realizar el cambio de contexto es tiempo no disponible para trabajo productivo. Un "planificador" determina qué hilo debe ejecutarse luego, basado en prioridades y si está listo para ejecutarse. Un kernel no preferente necesita que cada tarea haga algo explícito para darle control al

procesador de manera que el planificador pueda realizar el cambio de contexto. Un kernel preferente no tiene dicho requerimiento, en consecuencia, si un hilo se ejecuta durante demasiado tiempo se podría retirar la atención del procesador.

Los planificadores cooperativos trabajan bajo un concepto muy similar a un kernel no preferente, con una diferencia importante. Los hilos necesitan ejecutarse hasta su término cada vez, ya que no hay noción de cambio de contexto y reinicio.

En su forma más simple, los planificadores cooperativos tienen métodos ejecutados en algún tipo de orden. Esto se puede mejorar mediante la planificación de la ejecución agregando un marcador de tiempo (*time tick*) u otro mecanismo de control de tiempo. Luego se planifican los hilos basados en tiempos de ejecución asignados. El planificador mantiene una lista de hilos y cada vez que aparece un "marcador de tiempo" los tiempos de expiración se reducen. Cuando el temporizador del hilo expira, se pone a la cola para ejecución y el temporizador se recarga.

La ejecución de cada hilo es realizado fuera de la rutina del procesamiento del marcador, lo que significa que todos los hilos se ejecutan en el mismo contexto. Esto elimina la necesidad de un mecanismo de control de concurrencia, que también puede agregar tiempo de procesamiento.

3. Estructura de Kipu

Hay varias maneras de implementar un planificador cooperativo como este. Una de ellas es tener un conjunto simple de hilos para ser ejecutado y algún tipo de ciclo de tiempo para ser utilizado como un temporizador. Así es como se estructuró el proyecto original. Con el tiempo, Kipu fue actualizado y ahora está basado en un montículo (*heap*) que implementa una cola de prioridades.

Una cola de prioridades es una estructura de datos similar a una cola, pero cada elemento tiene una "prioridad". En una cola de prioridades, el elemento con la prioridad más alta se retira antes que cualquier otro elemento. Un montículo (*heap*) es una estructura de datos basada en árbol, donde el valor contenido en cada nodo es mayor o igual a los elementos de los hijos de ese nodo (Main, 2002). Así, el elemento con el mayor (o el menor) valor siempre está en la raíz y se retira en primer lugar. Esto proporciona una implementación eficiente de la cola de prioridades.

El Kipu original fue dividido en dos partes: procesamiento del marcador y procesamiento del hilo. Cuando era recibido un marcador, se procesaba una lista de hilos y se realizaba el decremento de los temporizadores. Cuando el temporizador del hilo llegaba a cero, se planificaba la ejecución del hilo. Una vez que el

procesamiento del marcador se completa, puede efectuarse el procesamiento del hilo. Los hilos que fueron planificados para su ejecución son ejecutados y pasan el parámetro añadido cuando el hilo fue creado. El Kipu original funciona razonablemente y es fácil de entender e implementar, pero el análisis de tiempo de ejecución era O(N). Esto no es un problema para muchas aplicaciones, pero es posible mejorarlo y Kipu se actualizó usando una cola de prioridades basada en montículo (*heap*), de tal manera que es posible conseguir un análisis de tiempo de ejecución de O(logN) (Main, 2002).

Implementado de esta manera, el hilo ubicado al tope del montículo (*heap*) siempre será el siguiente en ser ejecutado. Cada proceso recibe un tiempo de recarga, el cual determina la frecuencia con la que se ejecuta el hilo. Cuando se retira un hilo del montículo (*heap*), se ejecuta el hilo, y de ser necesario es planificado nuevamente. Entonces se calcula un nuevo delta dando un vistazo al siguiente elemento y es posible que dos hilos terminen con la misma prioridad. Para evitar que los procesos sean permanentemente bloqueados, el hilo más antiguo al tope del montículo (*heap*) siempre tiene prioridad sobre un nuevo elemento que ha sido planificado nuevamente. Ya que estamos usando una cola de prioridades no se necesitan marcadores y simplemente se establece un intervalo de espera para cuando el siguiente hilo se ejecute. Sin embargo, esto significa que se necesita un mecanismo de intervalo de espera, de tal manera que el marcador podría regresar al juego. En ambas versiones de Kipu cada hilo es su propio objeto, por lo que no hay problema en mantener los datos específicos del contexto. Cada vez que una tarea "espera", realmente la tarea regresa al planificador y siempre que el objeto sobreviva, se mantienen los datos.

4. Uso de Kipu

Para utilizar Kipu primero se necesita producir una clase que extienda la clase abstracta *KipuThread*, tal como se muestra a continuación:

```
public abstract class KipuThread {
    long reload = 1;
    ...
    public abstract void execute();
    public abstract int send (Object param);
    public abstract void setup(Object param);
}
```

Luego se instancia un grupo de objetos *KipuThread*. Cada *KipuThread* necesitará mínimamente la implementación de un método de ejecución (*execute*). Cada trabajador puede implementar opcionalmente un método de configuración para realizar las responsabilidades al inicio y un método de envío si se desea un método de comunicación entre hilos.

Cada objeto de *KipuThread* es planificado para su ejecución en su tiempo de recarga, el cual se debe establecer cuando el objeto es instanciado. No se permite un tiempo de recarga cero ya que básicamente simula un bucle estrecho.

En el archivo Kipu.java, el método *KipuMain* necesita ser personalizado para un objetivo particular. En la versión de *Android* y *Unix*, el siguiente *KipuMain*() es satisfactorio y funciona mediante la extracción de un hilo del montículo (*heap*), calculando un nuevo tiempo, ejecutando el hilo, calculando un nuevo tiempo de recarga para el hilo y volviéndolo a insertar en el montículo (*heap*).

```
public void kipuMain() throws Exception {
  KipuThread kipuThread = null;
  long reload = 0;
  long time = 0;
  while(true) {
  if (reload > 0)
      Thread.sleep(reload);
  kipuThread = this.extract();
  if (kipuThread == null)
    throw new Exception("...");
  time = time + reload;
  kipuThread.time = time + kipuThread.reload;
  kipuThread.execute();
  this.insert(kipuThread);
  reload = this.peek().time - time;
  }
}
```

Debe de crearse una clase que invoque *KipuMain*(), que cree los objetos *KipuThread* requeridos, y que los inserte en el montículo (*heap*).

4.1 Android

Android presenta una interesante dicotomía con respecto al tiempo real embebido; por un lado, tiene un sistema operativo *Unix* totalmente funcional que soporta

cualquier característica imaginable en el nivel de sistema operativo que se requiera, y por otro lado, es portátil y alimentado mediante una batería, por lo que cualquier cosa que se haga para minimizar el consumo de energía será en beneficio de la aplicación y el dispositivo.

4.1.1 Pruebas en *Android*

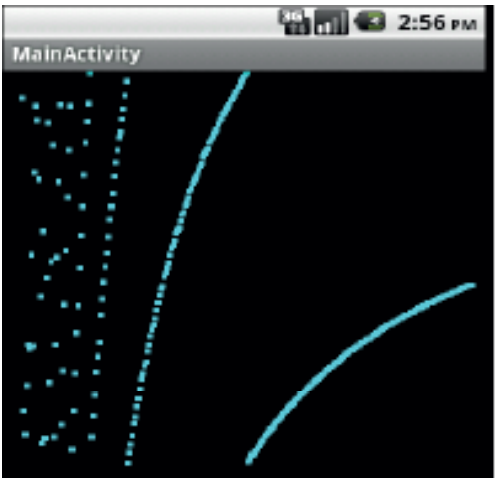
Se seleccionó *Android* para este proyecto ya que es fácil de crear objetivos de prueba para una comparación de desempeño paralelo, *Kipu* vs. *Runnable*. Cada objetivo tiene una clase llamada *AnimatedShape*, la cual tiene toda la información necesaria para dibujar un cuadrado geométrico y actualizar la información de ubicación, es decir, para mover el cuadrado. Durante la prueba simplemente se incrementó el valor de eje y (y-axis). También la clase *AnimatedShape* fue la clase seleccionada para extender la clase abstracta *KipuThread* para la implementación de Kipu. Para la aplicación abreviada del Kipu *AnimatedShape* no fue necesario extender la clase abstracta para *Runnable* y se prestó atención para asegurar que las implementaciones fueran lo más cercanas posibles a los objetivos.

```
public    class    AnimatedShape extends KipuThread {
    int   x,y,size,color;
    public    AnimatedShape(    int    _x,    int    _
        ...
  Rect getRect() {
    Rect rect =          new  Rect(x,y, x+size,y+size);
      return     rect;
    }
  public    void    execute(Object param) {
                y = (y+1)% 300;
  }
    ...
}
```

Cada objetivo (*Kipu* vs. *Runnable*) también tiene una clase creada llamada *Panel*, y esta contiene una colección de objetos *AnimatedShape*. Cuando se invoca *OnDraw* () la lista de formas es iterada y dibujada en sus posiciones apropiadas.

Para la prueba se crearon trescientos hilos de manera equivalente en ambos objetivos. Cuando se ejecutó la aplicación, el resultado fue la aparición de gráficos antiguos de baja resolución que salían en computadoras Apple o TRS-80, mostrando cuadrados que se desplazaban lentamente en la pantalla, tal como se muestra en la figura 1.

Figura 1. Resultado de ejecución de la aplicación



Elaboración propia.

4.1.2 Resultados de la prueba de *Android*

Para estimar la utilización del procesador (CPU) se invocó el *adb shell* del dispositivo del *Android* (*Android* 2.0, 550 Mhz TI OMAP 3430 ARM Cortex A8) y se ejecutó el programa UNIX *top*. Dos series de pruebas se llevaron a cabo con los trescientos hilos descritos anteriormente. En el primer conjunto de pruebas, la frecuencia de actualización se estableció en treinta milisegundos, en estas condiciones *Kipu* y *Runnable* estuvieron, consistentemente, dentro del 1 % de rapidez uno del otro. En el segundo conjunto de pruebas, la pantalla de frecuencia de actualización se estableció en sesenta milisegundos, y en estas condiciones Kipu fue casi 30 % más rápido que *Runnable*. Esto tiene sentido, ya que dibujar una pantalla requiere una gran intensidad del procesador (CPU), por lo que si una pantalla es dibujada más rápidamente consumirá más tiempo de procesador.

Estos resultados de las pruebas son interesantes por diferentes razones. En primer lugar, es importante señalar que Kipu siempre se mantiene a la par con respecto a lo construido con herramientas del sistema operativo (*Runnable*). Esto significa que es una consideración viable su uso en las implementaciones de JVM que no tienen soporte para hilos Java. Desde el punto de vista de *Android*, este estilo de sistema operativo podría ser una opción para los programadores de aplicaciones que encuentran problemas debido a la capacidad de respuesta del procesador. Otro aspecto de Kipu es que proporciona a los programadores con poca experiencia en el tema de concurrencia, una manera de crear aplicaciones multihilos sin tener que preocuparse por la sincronización. Finalmente, considera

alcanzar objetivos embebidos que no disponen de un mecanismo de soporte de hilos. El desempeño de hilos Kipu es suficiente para que un programador pueda implementar aplicaciones de multihilos de una manera habitual.

4.2 Investigación futura

La fase de pruebas debe ser ampliada para detectar problemas que normalmente se encuentran en sistemas operativos preferentes, tales como la sincronización. De manera alternativa, los sistemas operativos preferentes pueden tener la ventaja en las relaciones consumidor/productor donde un hilo se puede bloquear y esperar hasta que los datos estén disponibles antes de ejecutar. Ya que Kipu es estructurado, el hilo podría salir del estado *sleep* en tiempos establecidos y un sistema operativo preferente podría tener una ventaja al considerar este aspecto.

Finalmente, al explorar Source Forge uno puede encontrar una pequeña colección de implementaciones JVM, algunas de estas para entornos embebidos tales como Arduino, AVR y Lego Mindstorms. Muchas de estas implementaciones no soportan los hilos o incluso las excepciones de Java. Las pruebas futuras deberían incluir portar Kipu a una o más de estas plataformas para probar su viabilidad.

5. Conclusión

Kipu es práctico y fácil de entender como reemplazo de *Runnable* en muchas situaciones. Los requisitos subyacentes de Kipu a nivel de sistema operativo son mínimos y permiten una fácil portabilidad y despliegue en JVM embebidas. Ha sido útil en el aula y puede ser útil para proyectos relacionados con plataformas embebidas. Las pruebas han demostrado que el rendimiento es adecuado y Kipu debiera ser considerado para futuros proyectos.

Referencias

Labrosse, J. (1992). *MicroC/OS The real-time kernel*. Kansas, USA: R&D Publications.

Main, M. (2002). *Data structures and other objects using java* (2da ed.). Boston, USA: Addison-Wesley Longman.

Venners, B. (1999). *Inside the Java 2.0 Virtual Machine* (2da ed.). New York, USA: McGraw-Hill.

Kipu – Lightweight threads for Java

Michael Dorin*

Introduction

The introduction of Android has reinvigorated real-time, embedded programming in Java. While Oracle has their own Java Virtual Machine (JVM) designed for embedded devices, it is not always a practical choice. Furthermore several JVMs have become available for tiny devices. Most embedded JVMs do not try to support all java features and in many cases features like context switching, even if implemented, are overpowering for little devices.

In early 2007 I was introduced to a cooperative scheduler architecture in which tasks could be created and executed at different priorities. With cooperative scheduling, threads run until they complete or voluntarily release the CPU to other threads.

A 'C' implementation of the design material produced an extremely small cooperative operating system. This cooperative operating system requires all tasks run to completion but requires very little overhead.

As devices and tools have become more sophisticated, developers seemingly understand embedded and real-time programming less than ever before. Students may satisfactorily pass operating systems classes but still do not completely grasp concurrency and problems caused by concurrency. When they begin their professional lives they create applications built upon 'tapestries of threads' with little idea of how tugging one thread affects the reset of system. As a data structures instructor I realized I could demonstrate concurrency through combining data structures with a cooperative scheduler and Kipu was created. Kipu can be useful in an academic environment as well as low power java embedded devices, for example, NanoVM, which is designed for the Atmel AVR ATmega CPU, and does not support threads (http://sourceforge.net/projects/nanovm/, and http://www.harbaum.org/till/nanovm/ index.shtml) and even Android devices. Kipu can be build and run on most any version of java. It has been successfully run on current versions such as Java 8.

* I want to thank Scott McSpadden for his help with preparations for this project.

1. Java Threading review

It is not the intent of this paper to cover the internal details of exactly how threading in Java works as there is abundant information available. However some material on Java threads may be helpful to understand how Kipu can benefit some environments.

A thread is a 'thread of execution' of a program and standard Java allows multiple threads to be run concurrently (https://docs.oracle.com/javase/7/docs/api/ java/lang/Thread.html). The Java virtual machine specification defines a threading model that allows for a wide variety of threading architectures, including native threads. (Venners, 1999). Java threads additionally have their own working memory. With this in mind, one should remember the implemented thread mechanism might not be most efficient for every project.

Each thread is assigned a priority. Java can run threads at ten different priorities and there is no guarantee of time slicing. Threads with the highest priority are run first and threads with lower priorities run when the high priority threads complete or yield the CPU (Venners, 1999). Many people do not realize this and create situations of deadlock, priority inversion, and starvation.

Java supports thread synchronization using object locking and thread wait and notify. (https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html). Object locking is analogous to the 'enter critical region' function many RTOS's support and protects memory, objects, etc., which must not be modified by more than one thread at a time. Thread-wait and notify allows for producer consumer relationships among threads.

While Java threads are useful, too many threads can be a burden. After a while a programmer starts adding extra thread synchronization to fix problems they are encountering. The resulting program is often 'De facto cooperatively scheduled' (or worse) without being the developer's intention.

2. Cooperative Scheduling

When a multitasking operating system decides to run a different task/thread, it saves the current thread's context (i.e. the registers) and loads the context of the thread that should be run. This process is called context switching and adds overhead to the overall application (Labrosse, 1992). The time required to perform the context switch is time not available for productive work. A 'Scheduler' determines which thread should be run next, based on priorities and if it is ready to run. A *non-preemptive* kernel requires each task to do something explicitly to give up control of the CPU so the scheduler can perform the context switch. A preemptive kernel

has no such requirement consequently if a thread is running too long the CPU can be taken away.

Cooperative schedulers work on a concept very similar to a non-preemptive kernel, with one important difference. Threads need to run to completion each time, as there is no notion of *context switch and resume*.

Cooperative schedulers in their simplest form have methods executed in some sort of order. This can be improved by scheduling execution through adding a time tick or other timing mechanism. Threads are then scheduled based on assigned execution times. The scheduler maintains a list of threads and whenever a "time tick" occurs the expiration times are decremented. When a thread's timer expires, it is queued for execution and its timer is reloaded.

Execution of each thread is performed outside the tick processing routine, which means all threads run in the same context. This eliminates the need for concurrency control mechanism which also can add overhead.

3. Structure of Kipu

There are several ways a cooperative scheduler such as this can be implemented. One way is to have a simple array of threads to be executed and some kind of clock tick to be used for a timer. This is how the original project was setup. Overtime, Kipu was updated and now is based on a heap which implements a priority queue.

A priority queue is a data structure similar to a q**ueue,** but each element has a "priority". In a priority queue, the element with the highest priority is removed before any other elements. A heap is a tree-based data structure where the value contained in each node is greater than or equal to the elements of that node's children (Main, 2002). So the element with the greatest (or smallest) value is always in the root and is removed first. This gives an efficient implementation of a priority queue.

The original Kipu was broken into two parts: tick processing and thread processing. When a tick was received a list of threads is processed and required timers decremented. When a thread's timer reached zero, the thread was scheduled for execution.

After tick processing is complete, thread processing can be done. Threads that were scheduled for execution are executed and passed the parameter that was added when the thread was created.

The original Kipu works reasonably and is easy to understand and implement, but running time analysis was O(N). This is not a problem for many applications,

but is possible to do better and Kipu was updated to use a heap priority queue so is possible to have a running time analysis of O(log N) (Main, 2002).

Implemented this way, the thread at the top of the heap is always the one to be executed next. Each process is given a reload time which determines how often the thread is executed. When a thread is removed from the heap it is executed and, if necessary, is rescheduled. A new delta is then calculated by peeking at the next element and it is possible that two threads end up with the same priority. To avoid processes from being permanently blocked, the older thread on the top of the heap always has priority over a new or rescheduled element. Since we are using a priority queue no tick is required and we simply set a timeout for when the next thread should execute. This does mean you need a timeout mechanism however, so the tick might come back into play.

In both versions of Kipu each thread is its own object so there is no problem maintaining context specific data. Each time a task "yields" it is actually returning back to the scheduler and so long as the object survives, data is maintained.

4. Usage

To use Kipu one first needs to produce a class which extends the 'KipuThread' abstract class (Listing 1) then instantiate a group of 'KipuThread' objects. Each 'KipuThread' will minimally require implementation of an execute method. Each worker can optionally implement a setup method to do startup responsibilities and a send method if inter-thread communication method is desired.

```
public abstract class KipuThread {
   long reload = 1;
   ...
   public abstract void execute();
   public abstract int send (Object param);
   public abstract void setup(Object param);
}
```

Listing 1

Each KipuThread object is scheduled for execution at its reload time, which must be set when the object is instantiated. A reload time of zero is not allowed as that basically simulates a tight loop.

Within Kipu.java, the method kipuMain (Listing 2) needs to be customized for a particular target. For the Android and Unix version, the following kipuMain() is satisfactory and functions by extracting a thread from the heap, calculating a new time, executing the thread, calculating a new reload time for the thread and reinserting it back into the heap.

```
public void kipuMain() throws Exception {
  KipuThread kipuThread = null;
  long reload = 0;
  long time = 0;
  while(true) {
  if (reload > 0)
      Thread.sleep(reload);
  kipuThread = this.extract();
  if (kipuThread == null)
    throw new Exception("...");
  time = time + reload;
  kipuThread.time = time + kipuThread.reload;
  kipuThread.execute();
  this.insert(kipuThread);
  reload = this.peek().time - time;
  }
}
```

Listing 2

An encompassing class should be created to invoke kipuMain(), create the required KipuThread objects, and insert them into the heap.

4.1 Android

Android presents an interesting dichotomy with respect to embedded real-time. On the one hand it has a fully functional UNIX operating system supporting any imaginable OS feature required. On the other hand it is portable and battery powered so whatever we can do to minimize power consumption is to the benefit of the application and the device.

4.1.1 Android Testing

Android was selected for this project as it is easy to create test targets for a side by side performance comparison, 'Kipu' vs. 'Runnable'.

Each target has a class called AnimatedShape class which has all the necessary information drawing a geometric square and updating location information, i.e. moving it. During the testing it was simply incrementing the value for the y-axis. AnimatedShape was the class also selected to extend the abstract class KipuThread for Kipu implementation. See Listing 3 for an abbreviated implementation of the Kipu AnimatedShape. Extending the abstract class was not necessary for 'Runnable' and attention was paid to be sure the implementations were as close as possible for targets.

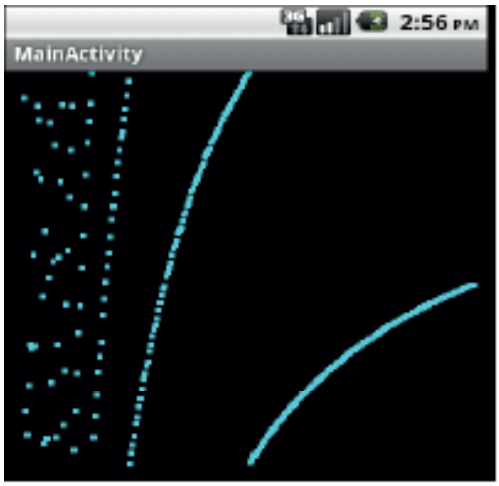
```
public class AnimatedShape extends KipuThread {
    int x,y,size,color;
    public AnimatedShape( int _x, int _
        ...
  Rect getRect() {
    Rect rect = new Rect(x,y, x+size,y+size);
      return rect;
    }
  public void execute(Object param) {
          y = (y+1)% 300;
  }
    ...
}
```

Listing 3

Targets have a Panel class created which contain a collection of AnimatedShape objects. When onDraw() is invoked the list of shapes is iterated through and drawn at their appropriate positions.

For the actual test three hundred threads were created in an equivalent manner on both targets. When executed the application gave the appearance of old low resolution graphics form an Apple or TRS-80 with squares slowly scrolling down the screen (see Image 1).

Image 1. Results



4.1.2 Android Test Results

To estimate CPU utilization 'adb shell' was invoked on the android device (Android 2.0, 550Mhz TI OMAP 3430 (ARM Cortex A8) and the UNIX utility 'top' was run. Two sets of tests were run with the 300 threads described above. In the first set of tests, the refresh rate was set to 30 milliseconds and Kipu and Runnable were consistently within 1% of each other. In the second batch of tests, the screen refresh rate was set to 60 milliseconds and Kipu was nearly 30% faster than Runnable. This makes sense, as drawing the screen is CPU intensive on this device. Thus drawing more rapidly alone will consume more CPU time.

These test results are interesting on several counts. First it is remarkable to note that Kipu always at least kept up with the built in operating system tools (Runnable). This means that it is a viable consideration for usage on JVM implementations that have no built in threading. From an Android standpoint this style of operating system could be an option for application developers who are constantly hitting the wall with CPU responsiveness. Another aspect of Kipu gives developers who have little experience with concurrency a manner to create multithreaded applications without having to worry about synchronization. Finally consider embedded targets, which do not have a threading mechanism build in. The performance of the Kipu threads is sufficient that a developer can implement their application in a typical manner.

4.2 Future research

Testing should be expanded to hit issues normally encountered in preemptive operating systems such as synchronization. Alternatively, preemptive operating systems may have the advantage in consumer/producer relationships where a thread can block and wait until data is available before running. As Kipu is structured, the thread would wake up at designated times and a preemptive operating system may have an advantage here.

Finally scanning Source Forge one can find a small collection of JVM implementations, some for embedded environments such as Arduino, AVR, and Lego Mindstorms. Many of these implementations do not support threading or even java exceptions. Future testing should include porting Kipu to one or more of these platforms to test viability.

5. Conclusion

Kipu is a practical easy to understand replacement for 'Runnable' in many situations. Underlying operating system requirements of Kipu are minimal and allow for easy porting and deployment to embedded java virtual machines. It has been useful in the classroom and can be useful for embedded projects. Tests have shown that performance is suitable and Kipu should be considered for future projects:

References

Labrosse, J. (1992). *MicroC/OS The real-time kernel*. Kansas, USA: R&D Publications.

Main, M. (2002). *Data structures and other objects using java* (2da ed.). Boston, USA: Addison-Wesley Longman.

Venners, B. (1999). *Inside the Java 2.0 Virtual Machine* (2da ed.). New York, USA: McGraw-Hill.