Entendiendo conceptos básicos de Threads con GTK#

por Martín A. O Márquez <xomalli@gmail.com>

Históricamente el soporte para las operaciones en paralelo u operaciones concurrentes en los lenguajes de programación no era común, la mayoría de lenguajes de programación proporcionaban dicho soporte como primitivas del sistema operativo que eran muy difíciles de programar y de mantener.

Fue hasta las décadas de los 70's y 80's del siglo XX que el departamento de defensa de los Estados Unidos construyo el lenguaje de programación Ada que entre sus capacidades permite a los programadores especificar la programación en paralelo.

Una de las técnicas mas populares para la programación concurrente son los hilos (Threads) esta técnica se basa en el principio de la multi-programación propia de los sistemas operativos modernos.

¿Qué es un Thread?

Un **thread** es la unidad básica de ejecución de un programa. En otras palabras, es la unidad más pequeña de ejecución para la cual un procesador reserva tiempo de procesador. Es otra forma de que varias actividades se ejecuten al mismo tiempo con el objetivo de paralelizar el código e incrementar el rendimiento de un programa, por eso a los threads se les conoce como procesos livianos (lightweight processes), ya que estos corren en paralelo como si fueran procesos aunque todos se ejecutan dentro de un mismo proceso que comparten recursos críticos tales como la memoria y el CPU. Un thread se compone de:

- Un Thread ID similar al Process ID.
- Un contador de programa.
- Un conjunto de registros.
- Una pila.

A diferencia de un proceso los hilos comparten recursos como la sección de datos, la sección de código, los descriptores de archivo o las señales entre otras.

Beneficios de la programación multithread

- El Multithreading (multihilado) permite a un proceso ejecutar múltiples tareas en paralelo. A cada tarea se le otorga su propio hilo de control, lo que ofrece los siguientes beneficios:
- **Rendimiento:** Porque todos los Threads se ejecutan dentro de un mismo proceso, el proceso no desperdicia recursos en hacer una copia de él mismo. Los costos de copiar procesos fork y de

ejecutar threads varían según el sistema operativo, aun así los threads son menos consumidores de recursos.

- **Simplicidad:** Los hilos son mucho más sencillos de programar y de mantener que los procesos.
- **Compartir la memoria global:** Como los hilos se ejecutan en un mismo proceso, cada hilo comparte el mismo espacio de direcciones de la memoria global del proceso.
- **Portabilidad:** Definitivamente los hilos son más portables que los procesos fork u otros mecanismos de programación concurrente.
- **Capacidad de respuesta:** Es posible que un hilo se mantenga haciendo una actividad, por ejemplo de descarga de un archivo mientras que otro puede continuar con una actividad de E/S, aquí se puede programar un esquema basado en el bloqueo de un hilo a nivel individual.

Estados de un Thread, su ciclo vital

Todo hilo debe cumplir con un ciclo de vida, este ciclo de vida comienza con un **Thread** en el estado **Unstarted** (inactivo) que es cuando se inicia, el hilo permanecerá en ese estado hasta que se invoque su método **Start** (Inicia) y así pasar al estado **Running** (en ejecución) que es cuando el sistema operativo le asigna un procesador, cuando esta en ejecución el hilo ejecuta todas las actividades que están en su delegado ThreadStart hasta terminarlas, una vez terminadas estas actividades pasa al estado **Stopped** (detenido) ya en este estado termina su ciclo, aunque un hilo puede pasar a este estado sin terminar sus actividades si se invoca su método **Abort** (abortar).

Otros estados para un hilo son:

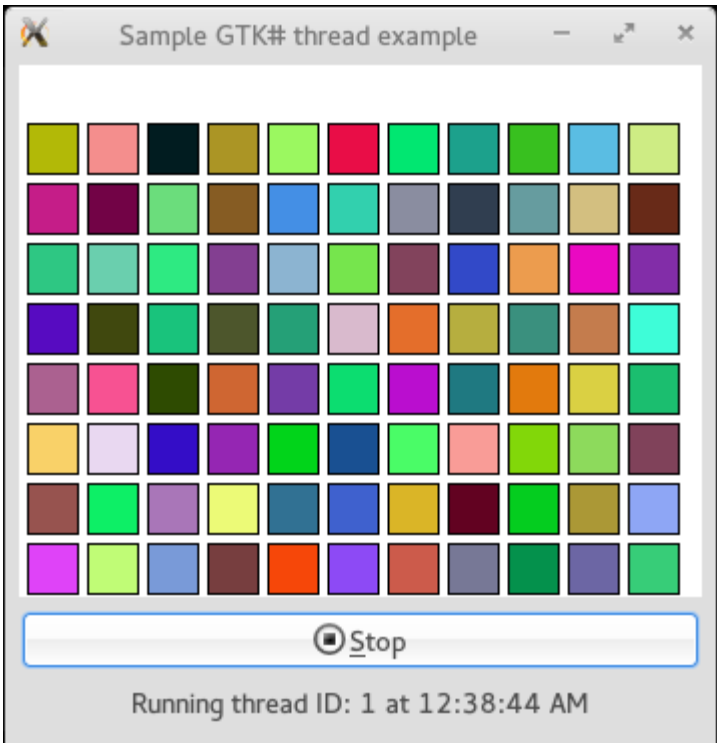
1. **Blocked** (Bloqueado) cuando el hilo tiene problemas en operaciones de E/S en donde necesita esperar por un recurso disponible.
2. **WaitSleepJoin** cuando se le pide al hilo que no se ejecute un determinado tiempo.
3. **Suspend** en este estado el hilo se interrumpe hasta que alguien invoque a su método Resume para regresar a la ejecución; en estos estados los hilos no pueden utilizar un procesador así se encuentre uno disponible.

Como nota adicional los métodos **Suspend** y **Resume** se encuentran obsoletos por lo que esta funcionalidad se implementa con la clase **Monitor**.

Para demostrar los conceptos básicos como la creación, el arranque y paro de un **Thread** he escrito el siguiente código como un ejemplo de como se escribe la creación de uno y la invocación de sus métodos **Start()** y **Abort()** respectivamente.

El siguiente programa es una sencilla animación en GTK# que dibuja rectángulos con un color diferente cada determinado tiempo utilizando las clases de dibujo de los ensamblados System.Drawing y gtk-dotnet. Este programa tiene dos botones: *play* y *stop*. El botón play inicia la animación en tanto que el botón stop detiene la animación.

Fig 1 Programa GTK# de una animación con un Thread.



El código fuente de este programa GTK# se divide en 3 clases:

- 1-. La clase program que es la clase inicial del programa que tiene al método Main(string[] args)
- 2-. La clase MainWindow que contiene la interfaz gráfica del programa aquí se crea el Thread y se controla su comportamiento por medio de los botones.
- 3-. La clase ColorBoxesCanvas que se encarga de dibujar las figuras de la animación.

Aquí el código fuente de la clase *Program.cs*

```
using System;
using Gtk;

namespace ColorBoxesThread
{
 class MainClass
 {
  public static void Main(string[] args)
  {
   Application.Init();
   MainWindow win = new MainWindow();
   win.Show();
   Console.WriteLine("GTK# Clock example running...");
   Application.Run();
  }
 }
}
```

Aquí el código fuente de la clase *MainWindow.cs*

```
using System;
using Gtk;
using System.Threading;

namespace ColorBoxesThread
{
 public class MainWindow : Gtk.Window
 {
  ColorBoxesCanvas canvas = null;
  Thread worker = null;
  VBox mainLayout = new VBox();
  HBox controlButtonsLayout = new HBox();
  Label lblStatusBar = new Label("Ready");
  Button[] controlButtons = {
   new Button("gtk-media-play"),
   new Button("gtk-media-stop")
  };

  public MainWindow() : base(Gtk.WindowType.Toplevel)
  {
   this.Title = "Sample GTK# thread example";
   this.SetDefaultSize(343, 311);
   this.DeleteEvent += new DeleteEventHandler(OnWindowDelete);
   this.BorderWidth = 6;
   canvas = new ColorBoxesCanvas();
   //Adjustment the controls
   mainLayout.BorderWidth = 1;
   mainLayout.Spacing = 6;

   controlButtonsLayout.Spacing = 3;
   controlButtonsLayout.BorderWidth = 1;
   controlButtonsLayout.Homogeneous = true;

   for (var i = 0; i < controlButtons.Length;i++)
   {
    controlButtons[i].CanFocus = true;
    controlButtons[i].UseStock = true;
   }
   //Setting the control panel

   controlButtons[0].Clicked += new EventHandler(Run);
   controlButtons[1].Clicked += new EventHandler(Abort);

   //Adding the button panel
   foreach (var button in controlButtons)
   {
    controlButtonsLayout.Add(button);
   }
   //Adding to the layout
   mainLayout.Add(canvas);
   mainLayout.Add(controlButtonsLayout);
   mainLayout.Add(lblStatusBar);
   this.Add(mainLayout);
   this.ShowAll();
  }

  protected void OnWindowDelete(object o, DeleteEventArgs args)
```

```
  {
   if (worker.IsAlive)
    worker.Abort();
   Application.Quit();
  }

  protected void Run(object sender,System.EventArgs e)
  {
   worker = new Thread(canvas.Repaint);
   canvas.KeepRunning = true;
   worker.Start();
   controlButtons[0].Visible = false;
   controlButtons[1].Visible = true;
   lblStatusBar.Text = "Running thread ID: " + Thread.CurrentThread.ManagedThreadId
    + " at " + DateTime.Now.ToLongTimeString();
  }

  protected void Abort(object sender, System.EventArgs e)
  {
   if (worker.IsAlive)
   {
    worker.Abort();
    controlButtons[0].Visible = true;
    controlButtons[1].Visible = false;
   }
   lblStatusBar.Text = "Stopping thread ID: " +
Thread.CurrentThread.ManagedThreadId
    + " at " + DateTime.Now.ToLongTimeString();
  }
 }
}
```

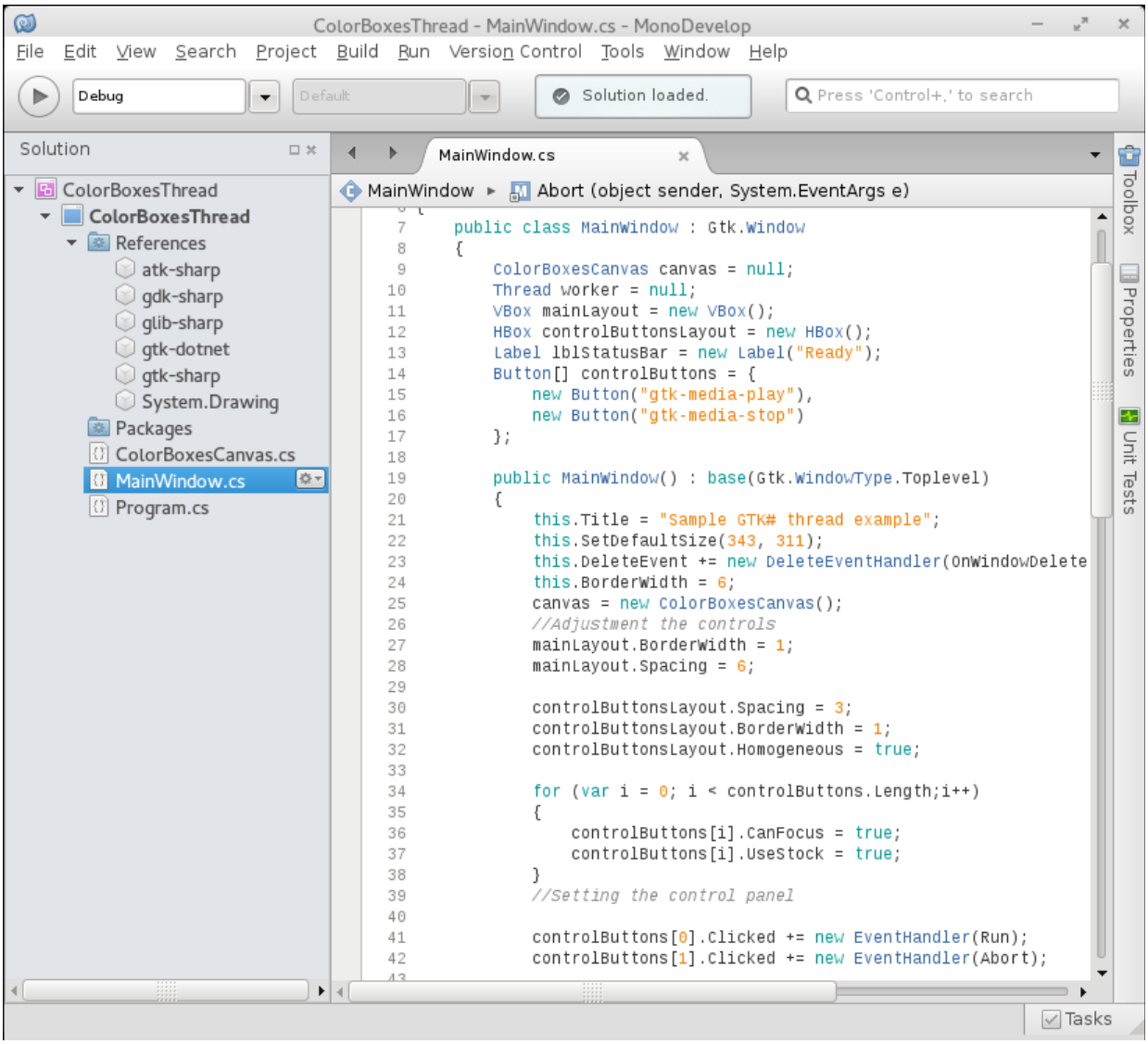
Aquí el código fuente de la clase *ColorBoxesCanvas.cs*

```
using System;
using Gtk;
using System.Drawing;

namespace ColorBoxesThread
{
 public class ColorBoxesCanvas : DrawingArea
 {
  public bool KeepRunning { set; get; }
  int sleepTime = 100;
  int red = 0,green = 0,blue = 0;
  Random random1 = new Random((int)DateTime.Now.Ticks & 0x0000FFFF);
  Random random2 = null;
  Random random3 = null;


  public ColorBoxesCanvas()
  {
   random2 = new Random((int)DateTime.Now.Ticks & 0x0000FFFF);
   SetSizeRequest(341,266);
   this.ExposeEvent += new ExposeEventHandler(ExposeHandler);
  }

  public void Repaint()
```

```
  {
   Console.WriteLine("Thread running...");
   while (KeepRunning)
   {
    this.ExposeEvent += new ExposeEventHandler(ExposeHandler);
    this.QueueDrawArea(0, 0, this.Allocation.Width, this.Allocation.Height);
    System.Threading.Thread.Sleep(sleepTime);
   }
  }

  protected void ExposeHandler(object o,ExposeEventArgs args)
  {
   random3 = new Random((int)DateTime.Now.Ticks & 0x0000FFFF);
   Gdk.EventExpose ev = args.Event;
   Gdk.Window window = ev.Window;
   using (Graphics g = Gtk.DotNet.Graphics.FromDrawable(window))
   {
    SolidBrush backBrush = new SolidBrush(Color.White);
    g.FillRectangle(backBrush,0,0,this.Allocation.Width,this.Allocation.Height);
    for (int j = 30; j < Allocation.Height - 15; j += 30)
    {
     for (int i = 5; i < Allocation.Width - 15; i += 30)
     {
      red = GetValue(random1);
      green = GetValue(random2);
      blue = GetValue(random3);
      SolidBrush foreBrush = new SolidBrush(Color.FromArgb(red,green,blue));
      g.FillRectangle(foreBrush,i, j, 25, 25);
      Pen forePen = new Pen(Color.Black);
      g.DrawRectangle(forePen,i - 1, j - 1, 25, 25);
     }

    }
   }
  }

  static int GetValue(Random r)
  {
   byte[] values = new byte[6];
   r.NextBytes(values);
   int index = new Random().Next(0, 6);
   int face = values[index];
   return face;
  }
 }
}
```

Para escribir el ejemplo creamos un proyecto en Monodevelop, creamos las clases, escribimos el código en cada clase correspondiente y compilamos. La solución debe verse como en la siguiente imagen:

Fig 2 Los archivos de la solución en Monodevelop.



Una vez compilada la solución la ejecutamos como se verá como en la siguiente imagen:

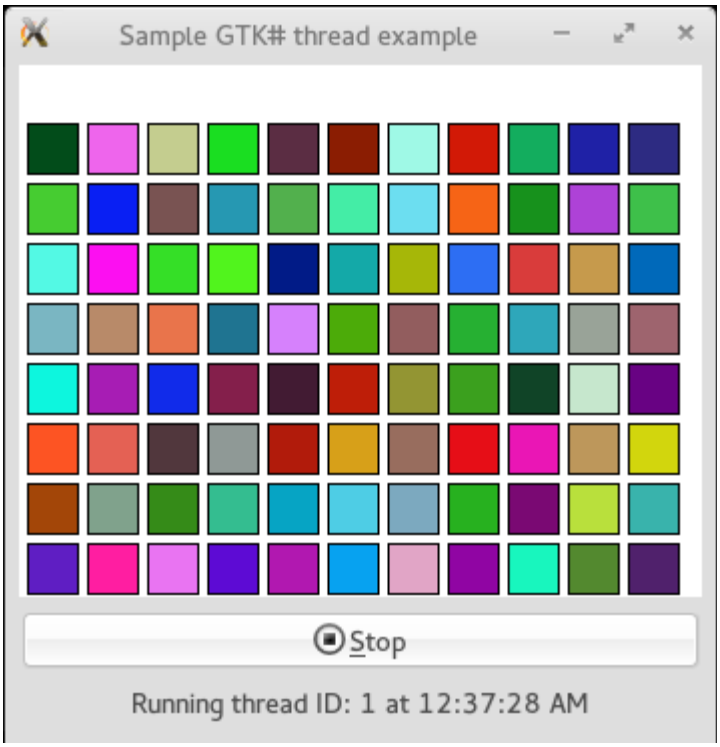
Fig 3 El programa al iniciar de su ejecución.

El programa muestra dos botones: play y stop, al oprimir el botón play se ejecuta el método **Run()**

```
protected void Run(object sender,System.EventArgs e)
 {
  worker = new Thread(canvas.Repaint);
  canvas.KeepRunning = true;
  worker.Start();
  controlButtons[0].Visible = false;
  controlButtons[1].Visible = true;
  lblStatusBar.Text = "Running thread ID: " + Thread.CurrentThread.ManagedThreadId
   + " at " + DateTime.Now.ToLongTimeString();
 }
```

Dentro de este método se crea un objeto **Thread** llamado worker al que se le pasa el delegado *canvas.Repaint* como argumento al constructor del objeto. Este delegado especifica la acción que realizará el subproceso durante su ciclo de vida, para este ejemplo el delegado es un método void que no recibe argumentos. Aquí el hilo permanece en estado **Unstarted** hasta que se llama a su método **Start()**, ejecutando este método el hilo pasa al estado **Running** y devuelve el control del programa al proceso que invoco al método **Start()**.

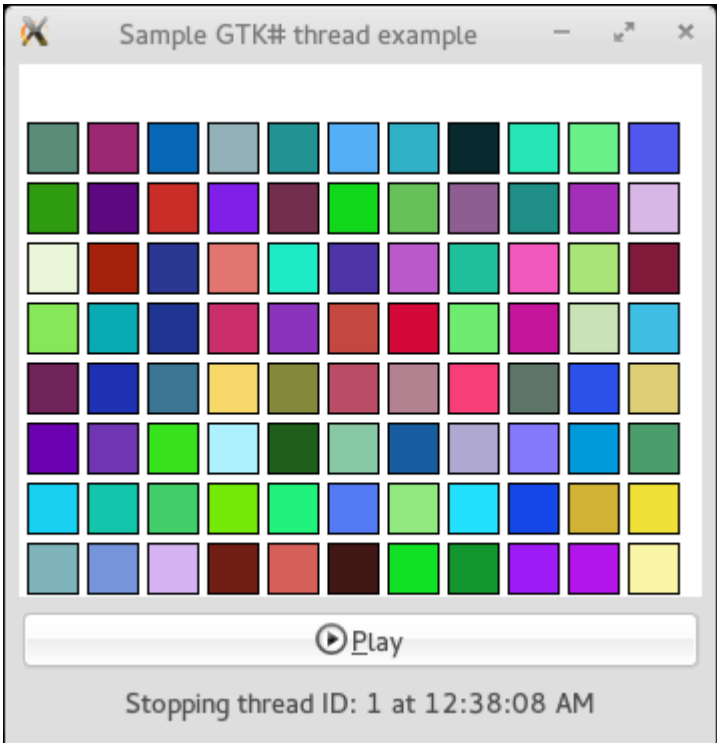
Fig 4 El programa al presionar el botón play y llamar al método Start().



Una vez en estado **running** el **Thread** puede pasar al estado **Stopped** o **Aborted** cuando termina la ejecución del delegado **ThreadStart**, esto indica que el subproceso terminado la tarea. Un hilo en ejecución puede forzar entrar al estado **Stopped** cuando se invoca al método **Abort()**, después de invocar este método el subproceso se detiene.

```
protected void Abort(object sender, System.EventArgs e)
 {
  if (worker.IsAlive)
  {
   worker.Abort();
   controlButtons[0].Visible = true;
   controlButtons[1].Visible = false;
  }
  lblStatusBar.Text = "Stopping thread ID: " +
Thread.CurrentThread.ManagedThreadId
   + " at " + DateTime.Now.ToLongTimeString();
 }
```

Fig 5 El programa al presionar el botón stop y llamar al método Abort()



Download el código fuente para Xamarin Studio o Visual Studio

Este documento está protegido bajo la licencia de documentación libre *Free Documentacion License* del Proyecto GNU, para consulta ver el sitio http://www.gnu.org/licenses/fdl.txt , toda persona que lo desee está autorizada a usar, copiar y modificar este documento según los puntos establecidos en la «Licencia FDL»