|  |
| --- |
|  |
| UD 2: Programación multiproceso (III) |
|  |
| Programación de servicios y procesos |



**Centro de estudios SEIM**

4 de septiembre de 2012

Autor: Daniel Miguel

UD 2: Programación multiproceso (III)

Programación de servicios y procesos

Programación multiproceso en c#

# La clase Process.

La clase System.Diagnostic.Process nos permite crear procesos, controlar en cierto modo su ejecución e incluso acceder a información acerca de los procesos que se estén ejecutando en el sistema (la misma información a la que se puede acceder a través del Administrador de Tareas de Windows).

|  |
| --- |
| Ejecución de procesos: **Método Start:** ejecuta el proceso de la misma manera que si estuviéramos invocando la orden shellExecute del sistema operativo. Ejemplo: Lanzamos InternetExplorer:  System.Diagnostics.Process.Start("iexplore.exe");  Lanzamos InternetExplorer para que abra una dirección:  System.Diagnostics.Process.Start("iexplore.exe","http://www.google.com");  **La clase** **ProcessStartInfo**  nos permite establecer la información para la ejecución de un proceso. Sus miembros son:   * Arguments: los argumentos que le pasamos al proceso. * FileName: nombre del ejecutable. * WindowStyle: el estilo de la ventana a mostrar. * CreateNoWindow: sirve para no mostrar una ventana al iniciar el proceso.  Ejemplo: ProcessStartInfo psi = new ProcessStartInfo();  psi.FileName="iexplore.exe";  psi.Arguments = "http://www.google.com";  Process.Start(psi);  Si al método Start le pasamos el nombre de un archivo, automáticamente abre el programa asociado con ese archivo, siempre y cuando ese tipo de archivo tenga un programa asociado. |

|  |
| --- |
| Detención de procesos: En una aplicación multiprocesos podemos invocar los siguientes métodos:   * Close: Libera todos los recursos del proceso. * Kill: Mata el proceso. * WaitForExit: Espera a que el proceso termine, se utiliza cuando queremos asegurarnos de que el proceso ha terminado antes de liberar sus recursos.  Ejemplo: Ejecutamos en segundo plano un comando de MSDOS.  Process proceso = new Process();  string salida = Application.StartupPath + "/output.txt";  string path = System.Environment.GetFolderPath  (Environment.SpecialFolder.System);  proceso.StartInfo.FileName = "cmd.exe";  proceso.StartInfo.Arguments =  "/C dir \""+path+"\" >> \"" + salida +"\" && exit";  proceso.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;  proceso.StartInfo.CreateNoWindow = true;  proceso.Start();  proceso.WaitForExit(1000); // 1 segundo de timeout  if (!proceso.HasExited) {  proceso.Kill(); // Finalizamos el proceso  } else {  MessageBox.Show("El proceso ha finalizado");  } |

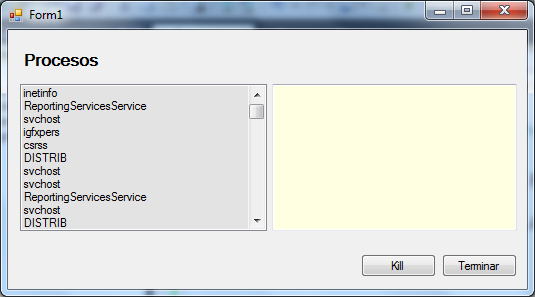
PRACTICA

# Crear proyecto:

Creamos un nuevo proyecto llamado Procesos.

# Dibujar formulario:

Dibujamos el siguiente formulario:



|  |  |  |  |
| --- | --- | --- | --- |
| CONTROL | TIPO | Propiedad | VALOR |
| frmProcess | Form | Text | Procesos |
| lstProcesos | ListBox | ScrollAlwaysVisible | True |
| btnSalir | Button | Text | Terminar |
| btnKill | Button | Text | Kill |
| txtInfo | TextBox | Text | “” |
| Multiline | true |

# Función llenar procesos:

|  |
| --- |
| private void LlenarProcesos()  {  procesos = Process.GetProcesses();  lstProcesos.Items.Clear();  foreach (Process p in procesos)  {  lstProcesos.Items.Add(p.ProcessName);  }  } |

# Evento load del formulario:

|  |
| --- |
| private void frmProcess\_Load(object sender, EventArgs e)  {  LlenarProcesos();  } |

# Doble click de la lista:

|  |
| --- |
| private void lstProcesos\_DoubleClick(object sender, EventArgs e)  {  try  {  i = lstProcesos.SelectedIndex;  String[] mensajes = {String.Format("Nombre de proceso:{0}\n", procesos[i].ProcessName),  String.Format("PID:{0}\n", procesos[i].Id),  String.Format("Prioridad: {0}\n", procesos[i].BasePriority),  String.Format("Uso de Memoria:{0}\n", procesos[i].WorkingSet64),  String.Format("Tiempo de CPU:{0}\n",procesos[i].TotalProcessorTime),  String.Format("Módulo principal:{0}\n",procesos[i].MainModule.FileName)};  textBox1.Lines = mensajes;  }  catch (Exception Ex)  {  MessageBox.Show(Ex.Message);  }  } |

# Botón Kill:

|  |
| --- |
| private void btnKill\_Click(object sender, EventArgs e)  {  DialogResult m;  m = MessageBox.Show("Está usted seguro?", "Advertencia", MessageBoxButtons.YesNo);  if (m == DialogResult.Yes)  {  procesos[i].Kill();  LlenarProcesos();  }    } |

Mecanismos de comunicación entre procesos

En ocasiones queremos que distintos procesos intercambien información. Guardar esta información en ficheros de disco no siempre es la mejor de las opciones. Los sistemas operativos suelen proporcionar unos mecanismos mejores, como es el caso de las tuberías o “pipes”.

Un programa estándar en Windows posee, por defecto, tres canales de entrada/salida: la entrada estándar StdIn, la salida estándar y por último la salida de error estándar (Stdin, stdout y stderr).

Poniendo *true* en las propiedades **RedirectStandarInput** y **RedirectStandarOutput** y **RedirectStandarError** de la Clase *ProcessStartInfo* podemos hacer que nos lleguen las entradas y salidas de otros procesos al nuestro. Esto solo funciona para procesos que utilicen como entrada y salida la consola.

# Pipes anónimos:

Un pipe anónimo consiste en un canal de comunicación que se establece entre dos procesos de la propia máquina, de tal manera que el servidor lo abre y los clientes pueden hacer uso del mismo.

Los pipes son unidireccionales y pueden ser de entrada o de salida. Los pipes de salida sirven para que el proceso servidor envíe información a los procesos clientes y los de entrada sirven para que el proceso servidor reciba información de los procesos clientes.

Los pipes anónimos permanecen mientras el proceso servidor los mantenga abiertos, y tiene que pasar de alguna manera su identificador a los procesos clientes, ya que dicho identificador es temporal. Esta es la principal diferencia con los pipes con nombre que los gestiona el S.O. y los procesos clientes pueden acceder a ellos con un nombre conocido.

Los pipes se asocian a un Stream que será de lectura (Reader) o de escritura (Writer) dependiendo del extremo de la comunicación que estemos implementando. Esto facilita enormemente la transferencia de información.

|  |
| --- |
| Ejemplo: El siguiente programa hace de servidor que crea un pipe y un subprocesos. Al crear el subprocesos le pasa el nombre del pipe para que lo utilice.  El proceso hijo lee teclas de teclado y lo escribe en el pipe, el proceso padre mostrará un msgbox con las letras introducidas: Proceso padre: static void Main(string[] args)  { using (AnonymousPipeServerStream apss = new AnonymousPipeServerStream(PipeDirection.In,HandleInheritability.Inheritable))  { Process hijo = new Process();  hijo.StartInfo.FileName = "..\\..\\..\\ProcesoHijo\\bin\\Debug\\ProcesoHijo.exe";  hijo.StartInfo.Arguments = apss.GetClientHandleAsString();  hijo.StartInfo.UseShellExecute = false;  hijo.Start();  apss.DisposeLocalCopyOfClientHandle();  StreamReader s = new StreamReader(apss);  char vble=' ';  vble = (char)s.Read();  String mensaje="";  while (vble!='\*')  {  mensaje += vble;  vble = (char)s.Read();  }  MessageBox.Show(mensaje);    }  }  } Proceso hijo: static void Main(string[] args)  {MessageBox.Show(args[0]);  using (AnonymousPipeClientStream apcs = new AnonymousPipeClientStream(PipeDirection.Out, args[0]))  {using (StreamWriter s = new StreamWriter(apcs))  { ConsoleKey x = ConsoleKey.A;  do  {  if (Console.KeyAvailable == true)  {  ConsoleKeyInfo cki = Console.ReadKey(true);  x = cki.Key;  s.Write(cki.KeyChar);  Console.WriteLine(cki.KeyChar);  }  } while (x != ConsoleKey.Escape);  s.Write('\*');  }  }  }  } |

# Pipes con nombre:

Un pipe con nombre es una canalización que gestiona el sistema operativo y que una vez abierta puede ser utilizada por cualquier proceso mediante el nombre de la misma. En algunos sistemas operativos (UNIX) también se conocen como FIFOs.

El servidor tiene que esperar las conexiones y los clientes tienen que indicar el nombre del servidor además del nombre del pipe.

|  |
| --- |
| Ejemplo Al cargarse un formulario el servidor crea un pipe de servidor con nombre y se queda a la espera de conexiones.  El cliente deja la información en el pipe y cuando en el servidor leemos el pipe recuperamos la información. Servidor: (Formulario del Servidor, el cliente es por consola).  public partial class Form1 : Form  {  NamedPipeServerStream npss;  public Form1()  {  InitializeComponent();  npss = new NamedPipeServerStream("pepepipe", PipeDirection.In);  npss.WaitForConnection();  }  private void button1\_Click(object sender, EventArgs e)  {    StreamReader sr = new StreamReader(npss);  String texto = sr.ReadToEnd();  textBox1.Text += texto;  sr.Close();      }  }  } Cliente: Se conecta al servidor local (nombre de servidor “.”) para utilizar la canalización “pepepipe”.  static void Main(string[] args)  {  NamedPipeClientStream npcs = new NamedPipeClientStream(".", "pepepipe", PipeDirection.Out);  npcs.Connect();  ConsoleKey k = ConsoleKey.NumPad0;  StreamWriter sw = new StreamWriter(npcs);    while (k != ConsoleKey.Enter)  {  if (Console.KeyAvailable)  {  ConsoleKeyInfo kki = Console.ReadKey();  k = kki.Key;  sw.Write(kki.KeyChar);  }  }  sw.Close();    }  } |

# Mensajes:

Los diferentes objetos del sistema operativo Windows se comunican a través de lo que llamamos mensajes. Estos mensajes es la implementación a bajo nivel del concepto de evento en los lenguajes de alto nivel como Java o .net.

Cuando los procesos tienen una ventana asociada pueden comunicarse mediante el envío y recepción de mensajes entre ventanas.

|  |
| --- |
| Funciones de la API para el envío y recepción de mensajes. **RegisterWindowMessage**: es una función del API que recibe como parámetro una cadena con el nombre del mensaje y devuelve un número (ID) de mensaje asociado.  **uso:**  //En primer lugar declaramos la función de la API, como función externa de la biblioteca user32.dll  [DllImport("user32")]  public static extern int RegisterWindowMessage(string message);  …  //En el momento de la invocación se recoge el código en una variable  int codigoMensaje = RegisterWindowMessage("WM\_MI\_MENSAJE");  **PostMessage:** es una function de la API que recibe 4 parámetros:   * El (ID) del destinatario del mensaje. Si usamos la constante HWND\_BROADCAST(0XFFFFFF) lo envía a todas las ventanas, aunque solo lo procesaran las que asocien procedimiento a dicho mensaje. Es de tipo IntPtr * El (ID) del mensaje a enviar. Puede ser el obtenido de la llamada a la función RegisterWindowMessage. Es de tipo int * Primer segmento de la información adicional al mensaje. Es de tipo IntPtr, si no queremos enviar información en este segmento pasaremos 0. * Segundo segmento de la información adicional al mensaje. Es de tipo IntPtr, si no queremos enviar información adicional en este segmento pasaremos 0.   **Uso:**  //Declaramos la función PostMessage de la librería user32.dll  [DllImport("user32")]  public static extern bool PostMessage(IntPtr hwnd, int msg, IntPtr wparam, IntPtr lparam);  …  //Llamamos a la función. Es un broadcast, sin información adicional.  PostMessage((IntPtr)0xffff, codigoMensaje, IntPtr.Zero, IntPtr.Zero); |

Para interceptar el mensaje redefiniremos el método WndProc en la clase Form de las ventanas que queramos que lo hagan.

|  |
| --- |
| protected override void WndProc(ref Message m)  {  if (m.Msg == idmensajeAP)  {  MessageBox.Show("Entra coche");  }  else if (m.Msg == idmensajeEX)  {  MessageBox.Show("Sale Coche");  }  else base.WndProc(ref m);  } |