

# Programando con TORO

*APIS para programadores.*

Por Matías E. Vara

Ultima modificación: 27 / 02 / 2008

## Sobre este Texto:

Se explicara que ideas debe tener el programador acerca de las reglas heurísticas que llevan al eficaz funcionamiento del sistema y la interfaz que ofrece **Toro** al programador.

Esta dirigido al usuario de la RTL y no explica nada acerca del funcionamiento interno, eso lo dejare para otro documento.

Es conveniente tener algún conocimiento acerca del lenguaje **Pascal** y en especial del Compilador **FreePascal**.

Un especial saludo a mi amigo **KW**,

Matías E. Vara,

[Toro.sourceforge.net](http://Toro.sourceforge.net)

[matiasvara@yahoo.com](mailto:matiasvara@yahoo.com) .

## Contenido

Multi-Threading.....	4
Memoria .....	7
Consola.....	10
Sistema de Archivo.....	12
Networking.....	15
Drivers.....	18
Ejemplo de aplicaciones de Usuario .....	19

## Multi-Threading

En **Toro** no existen procesos de usuario, solo existen threads de ejecución en modo núcleo. Se busca dividir un problema en miles de threads ejecutándose en forma paralela.

El primer thread en el sistema es la aplicación de usuario o *PASCALMAIN*.

El planificador implementa un modelo de thread cooperativo y permite la migración de threads entre procesadores sin la utilización de la instrucción "lock" que representa un verdadero cuello de botella para sistemas de multiprocesamiento.

La migración se realiza únicamente cuando el thread es creado , una vez creado en una CPU no puede migrar a otra.

El multiprocesamiento esta solo limitado por la constante *MAX\_CPU* declarada en el archivo *Arch.pas* de la correspondiente arquitectura. *CPU\_COUNT* contiene el numero de procesadores detectados.

La interfaz para la manipulación de threads es siempre la misma, el *ThreadManager* que provee **Freepascal** , las principales funciones son:

### BeginThread:

```
function BeginThread(SecurityAttributes: Pointer; StackSize: SizeUInt;  
ThreadFunction: TThreadFunc; Parameter: Pointer; CpuID: DWORD; var ThreadID:  
TThreadID): TThreadID;
```

Crea un thread de ejecución de la función dada en *ThreadFunction* en el procesador dado por *CpuID*.

### ThreadSwitch:

```
procedure ThreadSwitch;
```

Llamada al planificador , esta llamada es muy importante porque se le da el control al Sistema Operativo para que realice tareas de migración y otras rutinas y no se apropie de la CPU .

### EndThread:

```
Procedure EndThread(ExitCode: DWORD);
```

Llamada de Terminación del Thread, con el código de salida dado en *Exitcode* el cual deberá ser leído por el thread que lo creo.

Esta llamada no debe ser realizada por un procedimiento o función en el código de usuario, con solo realizar un *exit (code)*; el sistema operativo será el encargado de realizar la llamada.

## **WaitThread:**

*Procedure WaitThread(var Termination: LongInt; var ThreadID: TThreadID);*

Espera por la terminación de un thread. Solo la puede realizar el thread que lo crea. Retorna la causa de la terminación en *Termination* y el Identificador de Thread en *ThreadID*.

## **GetCurrentThreadId:**

*Function GetCurrentThreadId : TThreadID;*

Retorna el identificador del thread actual.

## **KillThread:**

*Function KillThread (ThreadID: TThreadID): DWORD;*

Mata al thread dado en *ThreadID*. Solo el thread que lo crea puede realizar esta llamada. Retorna 0 si se efectuó correctamente y -1 si no.

## **SuspendThread:**

*function SuspendThread(ThreadID: TThreadID): DWORD;*

Suspende la ejecución del thread dado en ThreadID, solo el thread que lo crea puede realizar esta llamada .Retorna 0 si se efectuó correctamente y -1 si no.

## **ResumeThread:**

*Function ResumeThread(ThreadID: TThreadID): DWORD;*

Reanuda la ejecución del Thread dado en ThreadID, solo el thread que los crea puede realizar esta llamada. Retorna 0 si se efectuó correctamente y -1 si no.

## **Sleep:**

*procedure Sleep(SleepTime: Int64);*

Duerme al proceso hasta que el registro *rdtsc* se incremente *SleepTime* veces.

Para la manipulación de secciones críticas por parte de las aplicaciones de usuario se proveen los siguientes procedimientos:

### **InitCriticalSection:**

```
procedure InitCriticalSection(var cs: TRTLCriticalSection);
```

Inicializa la sección crítica.

### **DoneCriticalSection:**

```
procedure DoneCriticalSection(var cs: TRTLCriticalSection);
```

Libera los recursos asociados a la sección crítica.

### **EnterCriticalSection:**

```
Procedure EnterCriticalSection(var cs: TRTLCriticalSection);
```

Cuando se retorna de este procedimiento, el thread que lo llama es el único corriendo el código ubicado entre *EnterCriticalSection* y *LeaveCriticalSection*.

### **LeaveCriticalSection:**

```
procedure LeaveCriticalSection(var cs: TRTLCriticalSection);
```

Luego de esta llamada el código protegido puede ser ejecutado por otro thread.

Pueden encontrar un descripción mas detallada acerca de estos procedimientos en el archivo *prog.pdf*, dentro de la documentación de **Freepascal**.

# Memoria

**Toro** implementa un modelo de memoria **NUMA** (Non-Uniform Memory Access). Cada procesador posee un banco de memoria asignado en la inicialización del sistema. Esto permite que la unidad de memoria no posea secciones críticas al realizar asignaciones y, con la nueva tecnología de Hipertransporte, que los procesadores acceden a ciertas regiones de memoria preestablecidas con mayor velocidad evitando posibles cuellos de botellas.

La Asignación por procesador -por ahora- es simple, solo se divide la memoria disponible por el número de procesadores en el sistema. Cuando se aloca memoria siempre se toma del procesador donde se está ejecutando el thread.

La versión para **i386** utilizaba un modelo de memoria plano sin implementar paginación, mientras que sobre **AMD x86-64** para entrar en el modo de 64 bits (**long mode**) necesariamente se debe activar la paginación, lo que hace **Toro** en la inicialización del sistema sobre esta arquitectura es mapear toda la memoria física en la misma dirección virtual y por lo tanto también se tiene un modelo de memoria plano, actualmente **Toro** puede manipular hasta 512 Gb de memoria.

El sistema trabaja con dos diferentes allocadores estos son:  
El **Global Allocator** y el **Local Allocator**.

## ***Global Allocator:***

La memoria que manipula es del tipo residente, por lo general son estructuras que son solicitadas y liberadas de forma continua. Son alocaiones que pueden ser manipuladas por muchos threads.

El *Cache Allocator* posee un cache de objetos que comienza con 32 bytes, 64,128... hasta 32 MB, la lista usa potencias de 2 y entre cada término genera 3 listas más distribuidas al 25% por lo que resulta el directorio: 32, 40, 48, 56, 64,80, 96, 112, 128... 32MB.

El programador debe saber esto con el fin de disminuir las pérdidas en la asignación y la posible fragmentación interna.

Los procedimientos para la alocaión y la liberación de memoria son:

## **GlobalAlloc:**

*Function GlobalAlloc (Size: PtrInt): Pointer;*

Quita un bloque de tamaño dado en *Size* del cache y devuelve su puntero.

## **GlobalFree:**

*Procedure GlobalFree (Size: PtrUInt; Address: Pointer);*

Devuelve al cache el bloque dado en *Address*, de tamaño dado en *Size*.

## **Local Allocator:**

Por otro lado, este alocador asigna memoria de tiempo de vida muy corto, que solo es utilizada por el thread que la solicita, como pueden ser cadenas de caracteres, parámetros, etc.

Se encarga de la captura de los procedimientos *GetMem* y *FreeMem*, que realiza el compilador y la aplicación de usuario.

El alocador funciona como una pila, asignando bloques de un tamaño establecido en `THREAD_ALLOCSIZE` y un puntero se incrementa de acuerdo al tamaño solicitado, por lo que la alocación es muy rápida, pero genera fragmentación interna.

La liberación de la memoria se produce únicamente cuando el thread termina de ejecutarse, y no la realiza el usuario sino el Sistema Operativo.

Los procedimientos para la asignación son implementados a través del *MemoryManager*:

## **GetMem:**

*Function GetMem (Size: PtrInt): Pointer;*

Devuelve el puntero a un bloque de memoria de tamaño dado por *Size*.

## **FreeMemSize:**

*Function FreeMemSize (P: Pointer; Size: PtrInt): PtrInt;*

Libera la memoria punteada en *P* de tamaño dado en *Size*.

## **AllocMemBlock:**

*Function AllocMem (Size: PtrInt): Pointer;*

Equivalente a *GetMem* pero el bloque devuelto está relleno de ceros

## **ReAllocMem:**

*Function ReAllocMem (var P: Pointer; OldSize, NewSize: PtrInt): Pointer;*

Expande el bloque dado en *P* de tamaño dado en *OldSize* y retorna un Nuevo bloque de tamaño dado en *NewSize*.

El programador debe saber que la memoria asignada con el procedimiento *GetMem* , al utilizar esta política de asignación , no es buena para bloques grandes y que son utilizados por muchos threads , porque una vez terminado el thread que lo solicito el bloque es liberado y puede ocasionar graves problemas si se sigue utilizando.

## ***Thread Local Storage:***

**Toro** soporta la declaración de variables globales, cuyo contenido es local para cada Thread (*Thread Local Storage*), la declaración de este tipo de variables es como siempre:

*Threadvar variable : longint;*

Esta memoria es manipulada por el **Global Allocator**.

## Consola

Los procedimientos involucrados en el manejo de la consola se encuentran en *Toro/rtl/Drivers/Console.pas* , estos son muy simples por el momento y solo permite acceder a la consola en modo texto. El modelo de planificación implementado hace que estos procedimientos no necesiten protección de accesos provenientes de la CPU Local , pero el programador debe saber que no hay protección de accesos provenientes de otras CPUS, lo cual no es grave para la escritura sobre la consola pero si para la lectura puesto que puede ocurrir que un thread permanezca dormido para siempre esperando una interrupción del teclado .  
Los procedimientos implementados son :

### WriteConsole:

```
procedure WriteConsole(Format: String; Args: array of PtrUInt);
```

Escribe una cadena de caracteres con formato , terminada en el carácter nulo . *Format* es una *String*.

Los caracteres con formato están antecidos de " %" y son :

`c' : Imprime el carácter ASCII correspondiente al argumento.

`h' : Imprime en formato Hexadecimal el argumento.

`d' : Imprime en formato decimal el argumento.

Los caracteres de control de la consola están antecidos de `\' y son:

`l' : Limpia la consola.

`n' : Genera un cambio de línea.

`v' : Imprime 9 caracteres en blanco.

### ReadConsole:

```
procedure ReadConsole(var ch: Char);
```

Lee un carácter de la consola .

### ReadConsoleLn:

```
procedure ReadLnConsole(Format: Pchar);
```

Lee una cadena de caracteres de la consola hasta que se presione Enter.

## **EnabledConsole and DisabledConsole:**

EnabledConsole habilita la escritura inmediata sobre la consola cuando se presiona una tecla y DisabledConsole lo deshabilita.

## Sistema de Archivo

El Sistema de Archivo es el encargado del acceso a bajo nivel a los dispositivos de almacenamiento de bloques (BlockDrivers) y al acceso de alto de nivel a los sistema de archivos montados en el sistema .

El acceso es implementado a través del **Virtual Filesystem** de **Toro**, que permite manejar cualquier tipo de Sistema de Archivo y dispositivo de almacenamiento masivo , ofreciendo una interfaz simple al diseñador de drivers .

Actualmente fueron desarrollados los drivers para Discos IDE y EXT2 FileSystem.

Los dispositivos de bloques son identificados con un Nombre (String) y un número denominado Menor (longint).

El acceso a los recursos del Sistema de Archivo no es uniforme, el programador de la aplicación debe dedicar el acceso a un dispositivo para un determinado procesador, solo ese procesador podrá acceder al dispositivo.

La llamada al sistema dedicada a esto es:

```
function DedicateBlockDriver(name: string;CPUID: longint): boolean;
```

Esta llamada dedica el Driver de Bloques en *name* al procesador en *CPUID*, y retorna *verdadero* si fue realizado correctamente.

El acceso a los dispositivos de bloque a bajo nivel es realizado por las llamadas al sistema:

### **SysOpenBlock:**

```
Function SysOpenBlock (Name: String;Minor: longint):THandle;
```

Devuelve un descriptor de archivo de bloque, del dispositivo de número dado por *Minor* perteneciente al driver *Name* , retorna *nil* si la operación no fue realizada correctamente.

### **SysReadBlock:**

```
function SysReadBlock(FI: THandle;Block,Count: longint;Buffer: pointer):longint;
```

Lee una cantidad *Count* de bloques a partir del bloque dado en *Block* del dispositivo en el descriptor de archivo *FI* y los copia al puntero *Buffer*.

El tamaño de los bloques dependerá del driver , este coincide con el tamaño físico y por lo general es de 512 bytes. Retorna el numero de bloques leídos

## **SysWriteBlock:**

*Function SysWriteBlock(Fl: THandle;Block,Count: longint;Buffer: pointer):longint;*

Copia a partir del bloque en *Block* una cantidad *Count* de bloques punteados por *Buffer* al dispositivo punteado por el descriptor *Fl*. Retorna el numero de bloques escritos.

Cuando se abre un archivo de bloque no necesita ser cerrado. El acceso siempre se realiza sin intervención del Buffer-Cache.

Para el acceso a alto nivel, es decir para acceder a un sistema de archivo primero se deberá Montar el sistema al CPU local. Cuando se monta un sistema este podrá ser accedido únicamente por el CPU local, puesto que todo recurso es dedicado.

Un sistema de archivo que reside en un dispositivo de bloque es montado con la llamada:

*Function SysMount (FileSystemName, BlockName: String; Minor: longint): longint;*

Monta el sistema de archivo del dispositivo *BlockName\Minor* utilizando el driver dado en *FileSystemName*. Este tiene el nombre del Driver de Sistema de Archivo que deberá estar instalado sobre el dispositivo para que se realice el montaje , un nombre puede ser "ext2" , "fat32" ,etc.

Por ahora se ha desarrollado el Driver para "ext2".

Una vez montado se podrán realizar las llamadas :

## **SysOpenFile:**

*function SysOpenFile(Path:pchar): THandle;*

Retorna el descriptor de archivo dado por la ruta en *path* o nil si falla.

## **SysSeekFile:**

*function SysSeekFile(FileDesc: THandle;Offset,Whence: longint): longint;*

Posiciona el descriptor de archivo en *Offset* utilizando el algoritmo dado en *Whence*. El algoritmo podra ser:

*SeekSet* coloca como nueva posición a *Offset*.  
*SeekCur* suma a la posición actual *Offset*.  
*SeekEof* se posiciona al final del archivo.

## **SysReadFile:**

*Function SysReadFile(FileDesc: THandle;count:longint;Buffer:pointer): longint;*

Lee del archivo dado en *FileDesc* una cantidad de caracteres dado por *count* y los copia al puntero dado por *Buffer*.  
Retorna el numero de bytes leídos.

## **SysWriteFile:**

*function SysWriteFile(FileDesc: THandle;count:longint;Buffer:pointer): longint;*

Escribe en el archivo dado por *FileDesc* , una cantidad de caracteres dado por *count* del puntero dado por *Buffer*.  
Retorna el numero de bytes escritos.

## **SysCloseFile:**

*procedure SysCloseFile(FileDesc: THandle);*

Cierra el archivo dado en *FileDesc* liberando todos los recursos .

El acceso a los dispositivos de bloque por el Sistema de Archivo se realiza a través del Buffer-Cache disminuyendo el acceso a los dispositivos físicos.

## Networking

**Toro** provee acceso a redes implementando soporte para tarjetas ethernet y el Stack TCP/IP . El acceso se realiza a través de los sockets .

Como el resto de los recursos en **Toro** , cada interfaz de red debe ser dedicada a un procesador y solo este podrá acceder .

La llamada al sistema encargada de esto es :

```
function DedicateNetwork(Name: string; IP, Gateway, Mask: array of byte): Boolean;
```

Siendo *Name* el nombre del driver. *Ip* la Ip de la maquina en la Red, *Gateway* la Ip del Router y *Mask* la mascara de Subred.

Es inicializada la Interfaz de red con los parámetros dados y se comienzan a recibir paquetes . La interfaz de red solo puede ser utilizada en la CPU donde fue ejecutada la llamada .

Las llamadas al sistema implementadas por **Toro** para el acceso a los sockets son similares a las primitivas para acceso a los sockets de berkeley:

### **SysSocket:**

```
function SysSocket(SocketType: LongInt): PSocket;
```

Retorna un puntero a una nueva estructura Socket o nil si la operación no pudo realizarse.

### **SysSocketBind:**

```
function SysSocketBind(Socket: PSocket; IPLocal, IPRemote: TIPAddress; LocalPort: longint): Boolean;
```

Llena los campos del Socket , esto no es necesario debido a que el usuario tiene acceso a la estructura del socket , pero se implementa por compatibilidad. *Iplocal* es ignorada

*Socket* es un puntero a una estructura Socket , *Ipremove* la Ip de la maquina remota y *LocalPort* el puerto local .

Retorna siempre Verdadero .

## **SysSocketListen:**

*function SysSocketListen(Socket: PSocket;Queuelen: longint): boolean;*

Prepara al Socket para recibir conexiones remotas. Siendo *Socket* un puntero a una estructura Socket y *Queuelen* la cantidad máxima de solicitudes de conexión en espera .

El socket es puesto en modo servidor .

## **SysSocketAccept:**

*function SysSocketAccept(Socket: PSocket): PSocket;*

Espera por una conexión remota y devuelve un puntero a una estructura Socket con la conexión al host remoto establecida . El socket devuelto se encuentra en modo cliente.

## **SysSocketSelect:**

*function SysSocketSelect(Socket:PSocket;TimeOut: longint):boolean;*

Duerme al thread solicitante a la espera de nuevos datos del host remoto , de un cambio en el estado del socket o si ha pasado una cantidad *TimeOut* de tiempo .

Retorno *true* si algún ocurrió algún evento o *false* si el cronometro a expirado.

## **SysSocketSend :**

*function SysSocketSend(Socket: PSocket;Addr:Pchar;AddrLen,Flags: longint): Longint;*

Envía datos al host remoto . Siendo *Socket* un puntero a una estructura Socket , *Addr* un puntero a los datos , *AddrLen* la cantidad de datos y *Flags* es ignorado en **Toro**. Retorna el numero de bytes enviados .

## **SysSocketRecv :**

*function SysSocketRecv(Socket: PSocket;Addr: Pchar;AddrLen,Flags: longint) : Longint;*

Lee datos del buffer de recepción y los copia al puntero dado en *Addr* una cantidad *AddrLen* de bytes . Retorna el numero de bytes leídos .

## **SysSocketConnect :**

*function SysSocketConnect(Socket: PSocket): boolean;*

Conecta el Socket al Host remoto y retorna *true* si la operación fue correcta.

## **SysSocketClose :**

*procedure SysSocketClose(Socket: PSocket);*

Cierra la conexión con el Host Remoto . Libera todos los recursos alojados en el Socket.

## **Drivers**

Los drivers se ubican en la carpeta *Toro/rtl/Drivers* y deberán ser añadidos a la aplicación de usuario como cualquier otra unidad .

### **IdeDisk:**

El archivo *Toro/rtl/Drivers/Idedisk.pas* contiene los drivers para discos IDE , soporta hasta 4 unidades y es un driver temporal hasta la implementación del driver para discos SATA , con soporte para 32 unidades.

Para realizar las operación de escritura y lectura sobre estos dispositivos , antes deben ser abiertos correctamente , pasare a explicar la nomenclatura utilizada :

*ATA0* y *ATA1*, corresponden a la Unidad Maestra y Secundaria IDE ,respectivamente .A su vez cada unidad soporta los números Menores (Minors) :

0 , que corresponde a la Unidad Maestra ; 1,2,3, y 4 que corresponden a las particiones Primarias de la Unidad Maestra; 5 , que corresponde a la unidad esclava;6,7,8 y 9 que corresponden a las particiones primarias de la Unidad esclava.

Estos datos son requeridos cuando se realiza la operación *SysOpenBlock*.

### **Ext2:**

El archivo *Toro/rtl/Drivers/Ext2.pas* contiene los drivers para el manejo de Sistemas de Archivo EXT2. Se encuentra registrado bajo el nombre de '*ext2*', y con este debe ser llamado cuando se realiza un montaje.

Por el momento solo el posible realizar operaciones de lectura sobre los ficheros , con soporte de hasta 4 Mb de Tamaño.

### **Ne2000:**

El archivo *Toro/rtl/Drivers/ne2000.pas* contiene los drivers para el manejo de Tarjetas de Red ethernet compatibles con el modelo ne2000 . Por el momento solo detecta una placa de red.

## Ejemplo de Aplicación de usuario

A la hora de comenzar a programar se tiene que tener en mente siempre como es el funcionamiento de **Toro** con el fin aumentar la eficacia y el aprovechamiento de los recursos.

Los paquetes incluyen un ejemplo de aplicación de usuario *toro.lpr*, compilada utilizando **Toro** .Aquí citare línea por línea la aplicación *toro.lpr* .

Archivo */toro.lpr*:

```
program Toro;
```

```
{ $IFDEF FPC }  
{ $mode delphi }  
{ $ENDIF }
```

Le dice al compilador que reconozca la sintaxis de Delphi.

```
uses  
  Kernel in 'rtl\Kernel.pas',  
  Process in 'rtl\Process.pas',  
  Memory in 'rtl\Memory.pas',  
  Errno in 'rtl\Errno.pas',  
  Debug in 'rtl\Debug.pas',  
  Arch in 'rtl\Arch.pas',  
  Filesystem in 'rtl\FileSystem.pas',  
  NetWork in 'rtl\Network.pas',  
  Console in 'rtl\Drivers\Console.pas',  
  IdeDisk in 'rtl\Drivers\IdeDisk.pas',  
  Ext2 in 'rtl\Drivers\Ext2.pas',  
  ne2000 in 'rtl\Drivers\ne2000.pas'
```

Es vital declarar de esta forma las unidades del kernel, este debe ser el encabezado de cualquier aplicación de usuario. El programador debe tener conocimiento de los drivers con el fin de declararlos en la aplicación de usuario.

```
const  
  Welcome: PChar = #13+'Aca Toro64'+#13;  
  Chau: PChar = #12#13 + 'Chau'+#13;  
  
// User Stack TCP-IP Configuration  
MaskIP: array[0..3] of byte = (255,255,255,0);  
Gateway: array[0..3] of byte = (192,100,200,1);  
LocalIP: array[0..3] of byte = (192,100,200,100);
```

Estos valores están relacionados con la configuración del Adaptador de red **OpenVPN**.

Esta función en forma de thread responde las conexiones entrantes

```
function Bienvenida(Param: Pointer): Int64;  
var  
  Socket: PSocket;  
begin  
  Socket := PSocket(Param);  
  SysSocketSend(Socket, Welcome, Length(Welcome), 0);
```

Se envía el mensaje de bienvenida al host remoto

```
// wait 500 ms  
SysSocketSelect(Socket, 500);
```

Se espera por algún evento y se establece un timer de 500 ms.

```
SysSocketSend(Socket, Chau, Length(Chau), 0);
```

Se envía el mensaje de saludo y la conexión es cerrada.

```
SysSocketClose(Socket);  
end;
```

```
var  
  ThreadId: QWORD;  
  ServerSocket, ClientSocket: TSocket;  
Begin
```

```
  DedicateBlockDriver('ATA0',0);
```

Se dedica la controladora Maestra IDE a la CPU 0, para poder luego realizar el montaje.

```
SysMount('ext2', 'ATA0', 6);
```

Se monta la unidad Maestra , el numero menor 6 corresponde a la partición primaria del disco maestro, utilizando el Sistema de Archivo EXT2.

```
DedicateNetwork('ne2000', LocalIP, Gateway, MaskIP);
```

Se inicializa la interfaz de red especificando La IP de la maquina , la puerta de entrada y la Mascara IP. Utilizando los drivers para ne2000.

```
WriteConsole('Service waiting for connection ...\n', []);  
ServerSocket := SysSocket(SOCKET_STREAM);  
ServerSocket^.Sourceport := 80;  
SysSocketListen(ServerSocket, 5);
```

Se crea una estructura Socket y se prepara para recibir conexiones remotas .

```
while True do
begin
    ClientSocket := SysSocketAccept(ServerSocket);
```

El thread es dormido esperando por una conexión remota .  
Cuando despierta , la comunicación con el host remoto se realiza a través de otro thread y se continua esperando por otra conexión.

```
    WriteConsole('Accepting client\n',[]);
    BeginThread(nil, 4096, Bienvenida, ClientSocket, 0, ThreadId);
end;
end.
```