

Programming with TORO

APIs for application programmers.

by Matías E. Vara

Last Modification: 27 / 02 / 2008

About this paper:

In this paper I will talk about the best practices of programming with **Toro** and the APIs provided by **Toro**.

This document is intended for programmers.

Prerequisites : basic knowledge of **Pascal** Language and **Freepascal Compiler**.

Greet to my friend **KW**,

Matías E. Vara,

toro.sourceforge.net

matiasvara@yahoo.com .

Content

Multi-Threading	4
Memory.....	7
Global Allocator:	7
Local Allocator:.....	8
Thread Local Storage (aka TLS):	9
Console	10
File System.....	11
Networking	14
Drivers	16
Example of User Application.....	17

Multi-Threading

In **Toro** there is only the concept of threads executing in kernel mode.

When an application is running, it is divided in multiple threads, and every thread is executed on specific/dedicated CPU in parallel.

The first thread of the system is the procedure *PASCALMAIN*.

The scheduler is based on the cooperative threading model and threads can migrate between processors without the need of any locking mechanism (no "lock" instruction), which is the primary bottleneck and provokes contention on other systems.

The migration process of threads happens only when the thread is created; once a thread created to run on CPU #0 it can't migrate to another CPU after while.

The max numbers of CPUs in the system is defined by the constant *MAX_CPU*, located in the unit *Arch.pas*. The global variable *CPU_COUNT* is the actual number of CPUs detected on the computer. The interface to interact with threads is the *ThreadManager*, the functions and procedures implemented in **Toro** are:

BeginThread:

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

Creates a thread with callback function *ThreadFunction*, to run on CPU *CpuID*.

ThreadSwitch:

```
procedure ThreadSwitch ;
```

Call the scheduler, this call is very important because the control is transferred to the Operating System to perform some runtimes and immigration operations.

EndThread:

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

This procedure is called when the thread ends, the *ExitCode* is returned to the creator of the thread waiting for it, DO NOT call this procedure in user program only perform *Exit(code)*; in the code of

the thread procedure, and the operating system will perform the internal call to EndThread.

WaitThread:

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

Waits for a specific thread to end, only the creator of the thread can perform this action.

Returns the *Termination* code and the *Thread identification*.

GetCurrentThreadId:

Function GetCurrentThreadId : TThreadID;

Returns the actual thread identification.

KillThread:

Function KillThread (ThreadID: TThreadID): DWORD;

Kills the thread with the identification in *ThreadID*, only the creator of the thread can perform this action.

Returns 0 if OK or -1 if NOT OK.

SuspendThread:

function SuspendThread(ThreadID: TThreadID): DWORD;

Suspends the execution of thread *ThreadID*, only the creator of the thread can perform this action.

Returns 0 if OK or -1 if NOT OK.

ResumeThread:

Function ResumeThread(ThreadID: TThreadID): DWORD;

Resumes the execution of thread *ThreadID*, only the creator of the thread can perform this operation.

Returns 0 if OK or -1 if NOT OK.

For the manipulation of critical section for user application, the ThreadManager has these procedures :

Sleep:

procedure Sleep(SleepTime: Int64);

Sleeps the process until that the *rdtsc* register is incremented of *SleepTime* times.

InitCriticalSection:

procedure InitCriticalSection(var cs: TRTLCriticalSection);

Initialization of Critical Section.

DoneCriticalSection:

procedure DoneCriticalSection(var cs: TRTLCriticalSection);

Frees the resources of critical section.

EnterCriticalSection:

Procedure EnterCriticalSection(var cs: TRTLCriticalSection);

When this call returns, the calling thread is the only thread running the code between the *EnterCriticalSection* call and the following *LeaveCriticalSection* call.

LeaveCriticalSection:

procedure LeaveCriticalSection(var cs: TRTLCriticalSection);

Signals that the protected code can be executed by other threads.

For more information see the file *prog.pdf* in the documentation of **Freepascal**.

Memory

Toro implements **NUMA** memory model (Non-Uniform Memory Access).

Every CPU has a dedicated amount of memory, this amount is assigned during the system initialization. The memory manager doesn't use any critical section (no lock), and with the new AMD HyperTransport technology the processor can access some regions of memory very fast, and avoid bottleneck in memory access.

The memory manager always addresses the blocks of memory dedicated to the processor on which the thread is running.

In **i386** is a flat memory model without any pagination, while in **AMD x86-64** for jump to 64 bits (**long mode**) **Toro** must enable the pagination but all physical memory is mapped in same virtual address and the result is a flat memory model again. The latest version can support up to 512 GB of memory addressing (this limit can easily be extended through a parameter and a recompilation).

Toro has two memory allocators (2 memory manager):

The **Global Allocator** and the **Local Allocator**.

Global Allocator:

Allocate resident memory and "persistent" data. This type of memory can be shared with other threads running on the same CPU with no need for locking (due to the cooperative threading model design of Toro).

The Global Allocator has a directory of objects that ranges in 32 bytes, 64, 128 ... to 32 MB, the list uses the power of 2 and between two entries it has three additional lists distributed by 25%, the result is the directory:

32, 40, 48, 56, 64, 80, 96, 112, 128... 32MB.

The programmer must be aware of this internal design in order to optimize the allocation of memory and reduce the internal fragmentation.

The procedures for users are:

GlobalAlloc:

Function GlobalAlloc(Size: PtrInt): Pointer;

Allocates a block of memory from global pool and returns the pointer address.

GlobalFree:

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

Returns the block at *Address* to the Global Allocator memory pool.

Local Allocator:

This allocator works with memory that has short lifetime and it is used primarily for threads handling http requests, for example allocation of strings, parameters, etc.

The functions *GetMem* and *FreeMem* implements internally calls to the *LocalAlloc* and *LocalFree* functions of this local allocator.

It works like a stack, it originally assigns one block of `THREAD_ALLOCSIZE` bytes (1MB), this allocation is very fast but has internal fragmentation.

The release of memory happens when the thread ends.

The procedures for users are implemented using the *MemoryManager*:

GetMem:

Function GetMem (Size: PtrInt): Pointer;

Returns a pointer address of free block of memory with *Size* bytes.

FreeMemSize:

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

Free a memory block pointer *P* with *Size* bytes .

AllocMemBlock:

Function AllocMem (Size: PtrInt): Pointer;

Similar to *getmem*, the allocated memory content will be filled with zeroes.

ReAllocMem:

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

Should allocate a memory block *Size* bytes large, and should fill it with the contents of the memory block pointed to by *P*, truncating this to the new size needed. After that, the memory pointed to by *P* may be deallocated. The return value is a pointer to the new memory block. Note that *P* may be **Nil**, in which case the behavior is equivalent to *GetMem*.

Note: the programmer must be fully aware that the procedure *GetMem* works with this policy, it is not designed to allocate large blocks and returned memory pointers are not meant to be used by other threads, because the blocks of memory are released when the thread terminates.

Thread Local Storage (aka TLS):

Toro supports the declaration of threadvars. Threadvar are like global variable but their content is specific for the scope of the running thread. Declaration in Freepascal looks like:

```
threadvar variable : longint;
```

This memory is manipulated by the **Global Allocator**.

Console

The procedures to output to the Console are located in the file *Toro/rtl/Drivers/Console.pas*. These procedures are very simple to output content in text mode (mainly to track that Toro has booted successfully).

This procedure doesn't need protection from local CPU but there is no protection implemented when accessed by threads running on other CPUs. For write operations this is not very important, for read operations this may become an issue, because one thread can wait forever for a key interruption.

The procedures to access the console are :

WriteConsole:

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

Writes a null terminated string or Pascal string onto the screen with formatting options. The character's format preceding the format operation is "%":

'c' : Prints in ASCII code the argument.

'h' : Prints argument in Hexadecimal .

'd' : Prints argument in Decimal.

Console's control character is with "/":

'|' : cleans the console.

'n' : changes the line.

'v' : Writes nine black characters.

ReadConsole:

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

Read a character from console .

ReadConsoleLn:

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

Read a line from console until ENTER key is pushed .

EnabledConsole and DisabledConsole:

When a key is pushed ,Enable or Disable the writes of key to the screen .

File System

Through the File system the user can access to low level block devices and high level to File System mounted.

The Virtual File System implemented can manipulate any blocks devices or File System with similar interface for the programmer. Actually **Toro** has drivers to access IDE Disk and EXT2 File system. The Block devices are accessed using a name (String) and a Minor number(longint).

To access the blocks devices, it is not uniform, the user must dedicate devices to specific processor, and then only the threads running on this processor can access to the device. This design is intended to limit competition from multiple threads to access a same device.

The System call to dedicate a device is:

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

This call dedicates the device in *name* to the processor in *CPUID*, and returns true on success.

The System call to access block devices at low level is :

SysOpenBlock:

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

Returns a file descriptor of the driver from *Name* of the device number *Minor*. Returns nil on error.

SysReadBlock:

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

Reads *Count* blocks, starts at offset *Block* from *FI* descriptor file and copy the content into *Buffer*. The size of a block depends on the type of driver, it is usually equal to the underlying physical block, this is in general 512 bytes.

Returns the number of blocks read.

SysWriteBlock:

```
Function SysWriteBlock(FI: Thandle;Block,Count: longint;Buffer: pointer):longint;
```

Writes *Count* blocks, starts at offset *Block*, from *Buffer* to *FI* file descriptor. Return the number of blocks written.

Once a Block device is open, it doesn't need to be closed. Block read/write operations are not under the control of cache buffering.

To access files through the file system level, you first need to mount the file system to local CPU. When a file system is mounted only threads running on local CPU can access to it, because all resources must be accessed through a dedicated CPU.

A file system is mounted using the system call :

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

Mount the file system in *BlockName\Minor* using the driver *FileSystemName*, this latest value can be "ext2", "ext3", "fat", etc . Available file system at this time is "ext2".

Once the file system is mounted, the system calls to access it are:

SysOpenFile:

function SysOpenFile(Path:pchar): THandle;

Returns a file descriptor of file *path* or nil if fails.

SysSeekFile:

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

Moves the cursor of file descriptor in *FileDesc* at *Offset* using the *Whence* algorithm . Whence can be :

SeekSet put the cursor to *Offset* .

SeekCur increment *Offset* from current position .

SeekEof put the position to end of file.

SysReadFile:

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

Reads from *FileDesc* file descriptor, *count* bytes into *Buffer* pointer. Returns the number of bytes read.

SysWriteFile:

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

Writes to *FileDesc* file descriptor, a *count* bytes from *Buffer* pointer .
Return the number of bytes written.

SysCloseFile:

procedure SysCloseFile(FileDesc: THandle);

Closes *FileDesc* file descriptor and frees all resources.

Accessing blocks through File System APIs is performed through cache buffering in order to reduce physical access to the devices, and to globally improve performance.

Networking

Toro provides standard network features by supporting ethernet card and implementing TCP/IP Protocol. For programmers, interacting with the network is performed through socket APIs.

Every NIC (Network Interface Card) is dedicated to one processor and only this CPU can then access to the resource .

The system call to dedicate a network interface is :

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

Where *Name* is the name of the driver. *Ip* the IP of the machine in the network, *Gateway* the IP of the Router and *Mask* the subnet mask.

The access to socket in **Toro** is similar to the standard Berkeley APIs.

SysSocket:

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

Returns a pointer to Socket structure or nil if it fails.

SysSocketBind:

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

Fill the values in Socket structure

Socket is a pointer to a Socket record, *IPRemote* is the IP address of the remote machine and *LocalPort* the local port used in the connection. *IPLocal* is ignored.

Return always True.

SysSocketListen:

```
function SysSocketListen(Socket: PSocket; QueueLen: longint): boolean;
```

Prepares the Socket to accept remote connections. Where *Socket* is a pointer to Socket structure and *QueueLen* is a max number of client waiting for connection. In this case, the socket will be setup in server mode.

SysSocketAccept:

function SysSocketAccept(Socket: PSocket): PSocket;

Waits for remote connection and returns a pointer to remote socket when connection is established.

SysSocketSelect:

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

Sleeps current thread, waiting for new data from remote host, for any change in socket state or returns once the timeout has expired. Return True if any event happens and false on timeout expiration.

SysSocketSend :

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

Send data to Remote Host, *AddrLen* bytes from *Addr* pointer. **Toro** is currently ignoring *Flags*. Returns the number of bytes sent.

SysSocketRecv :

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

Read from remote socket and write received data to *Addr* pointer a number of *AddrLen* bytes. *Flags* are ignored. Returns the number of bytes read.

SysSocketConnect :

function SysSocketConnect(Socket: PSocket): boolean;

Connect to Remote Host and returns True when the operation has succeeded.

SysSocketClose :

procedure SysSocketClose(Socket: PSocket);

Close connection to remote host and free all resources allocated in *Socket*.

Drivers

This section will describe how are written existing drivers in **Toro**. The source code for drivers is located in *Toro/rtl/Drivers* and must be declared in the application like any other unit.

IdeDisk:

The source file of the IdeDisk driver is located in *Toro/rtl/Drivers/Idedisk.pas*, it supports up to 4 disks and is a temporary driver until the implementation of SATA disk driver is implemented, which will support up to 32 disks.

The nomenclature name to address the device is:

ATA0 and ATA1, meaning Master and Slave controller.

Every controller has these Minor numbers:

0 to address the master disk ; 1,2,3 and 4 are for primary partitions of the master disk; 5 to address the slave disk; 6,7,8 and 9 are for primary partitions of this slave disk.

This information is required when calling *SysOpenBlock*.

Ext2:

The file *Toro/rtl/Drivers/Ext2.pas* contains the code for EXT2 file system. Registered under the name "ext2", this is the name to use in all Mount related procedures. EXT2 driver is read-only at this time, with support up to 4MB file size.

NE2000:

The file *Toro/rtl/Drivers/ne2000.pas* has the drivers to handle ne2000 compatible ethernet card. At this time Toro detects only one NIC (Network Interface Card).

Example of User Application

When you begin a new program, you must know the policies of **Toro** in order to optimize access and concurrency to resources on your server.

The package includes one example of user application compiled by **Toro**. This section will describe the program a sample *Toro.lpr* on line at a time.

File */toro.lpr*:

```
program Toro;
```

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

Directive for the compiler, note that the code is compliant to be compiled also with the Delphi compiler.

```
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'
```

The units of the kernel must always be declared in the header of the user application and compiled with the application – Remember that the compilation process is generating a single binary including the kernel of Toro. The programmer must know what drivers to use and declare these units in the header.

```
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);
```

These values depend on the configuration you have chosen in the **OpenVPN** Network Adapter.

This function replies to remote connections:

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

Welcome message is sent to remote host.

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

The thread waits for an event from the socket and a timer is established.

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

Exit message is sent once the connection is closed.

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

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

Master IDE Controller is dedicated to CPU #0.

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

Master IDE Controller , First Primary Partition (Minor 6) is mounted using EXT2 driver.

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

The Network Interface is initialized and the parameters are configured using the ne2000 driver.

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

Socket structure is created and is prepared to receive external connections.

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

The thread sleeps for incoming new connection. At every new connection a new thread is created to process every request.

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