The Toro Project, a Pascal kernel

Introduction

This article introduces a kernel named Toro which I have been developing over the past 15 years. Put simply, a kernel is a program that enables another program (such as a user application) to create processes, to allocate and free memory, to access connected devices, and do various other activities. Toro is a kernel which offers OS-like services, implemented as a collection of Free Pascal units. Toro differs from other general-purpose kernels in that it is application-oriented, meaning that it is optimized to run only one application. This one application runs in isolation in the Toro system and can use any or all of the system resources without any interference.
In what follows, the first section outlines the history of the Toro project's development. A second section describes the kernel's architecture, and offers a comparison of Toro with more general-purpose kernels. A third section presents what a Toro application looks like. The concluding section gives an overview of ongoing work and future Toro development.

Toro History

I started Toro in 2003 while I was still at high school, and my main motivation was to learn more about Operating Systems. I spent six months doing research about Operating Systems and the x86 architecture. I read the Minix book, and Stalling's book about modern operating systems. Then, when I felt ready, I started to code. At that point I was learning to program using the Free Pascal compiler, so I decided to use Pascal to write the whole kernel. Without exception my colleagues considered this laughable. I remember a teacher at the Universidad Nacional de La Plata, Argentina, telling me that it was not possible to write an Operating System in Pascal. I think she had never actually tried to do it herself. In spite of this I started to code in 2003 and released the first version in 2004. This was a very simple version that could only boot from a floppy disk. It included a bootloader, support for paging memory, support for multitasking and a simple file system. I also developed various user applications such as *sh*, *echo*, and *ls* (see http://torokerneleng.blogspot.com.es/2010/11/old-toro.html?m=0 for more information).
2005 saw the release of the next version. This included several improvements: grub now replaced my own bootloader; the file system was completely rewritten as a virtual file system very similar to that in Linux; and I developed a driver for FAT12.
In 2006, I completely changed the direction of the project, focusing on how to improve the architecture of a kernel in which only one application will be executed. Such a kernel would scale well with respect to multi-core, and would compile within the user application. In addition, it would be able to run in different hypervisors, very much like containers nowadays. The result of these developments is the Toro Kernel that I describe in summary in the next section.

The Toro Kernel

The Toro kernel architecture is designed to improve the execution of only one user application, for instance a web server, a data server, or indeed any application that consumes a lot of resources and needs therefore to be executed alone in a system. These requirements shaped specific design improvements to the scheduler, to memory access routines and to the file and networking systems. To exploit all these improvements any user application needs to be specified in a particular way. The following description summarises these particular specifications.

In Toro, the scheduler is implemented via a cooperative algorithm. As each thread finishes its task it invokes the scheduler. The scheduler selects the first remaining thread that is in a ready state, and executes it. This thread executes until it decides to invoke the kernel again. Toro's scheduler does not need be executed repeatedly to reschedule a new thread, which is a major difference from other scheduling algorithms. In this way Toro avoids using an interrupt to count time, and so avoids the expensive context switching that other interrupt-based scheduling algorithms require. Additionally, both the thread and the kernel execute in supervision mode (i.e. protected mode in x86 architectures), thus making any context switching very lightweight.

In Toro, each resource is linked to a specific core. It is the user's responsibility to dedicate each resource to the appropriate core, whatever system resource is involved. For example, consider the system's memory. In Toro, the memory is linked or dedicated. The physical memory is divided by the number of cores. In a 2-core system with 512 MB of RAM, each core would have 256 MB. The main benefit of this memory dedication is that the allocation and the release of memory no longer need any synchronization between cores. This scales very well when the number of cores increases. Figure 1 illustrates the allocation of memory for a system with three cores. When a thread in Core1 requests memory, the allocator only assigns memory from Region 1. The allocator assigns memory depending on the core that executed the request. The allocators assign and release memory without exchanging information with the other allocators in other cores thus avoiding the need for any synchronization mechanism.

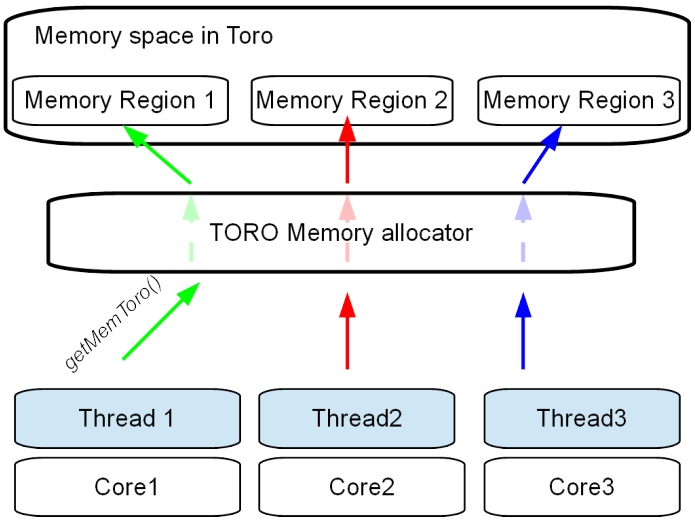

Figure 1. Dedication of Memory.

In an analogous way, file system and network resources are also dedicated by core. For a file system, the user must first dedicate the disk that contains the file system to one of the cores, and then dedicate the file system itself to the same core. For example, if we have a disk with an Ext2 file system and we dedicate it to core 0, the user may dedicate the Ext2 partition only to core 0. In the case of networking, the user must first dedicate a network card to a core before being able to dedicate the Networking to that core. Figure 2 illustrates the dedication of resources for a system with three cores, two disks and one network card. Each device is dedicated to a different core. Only the core to which the device is dedicated may access the dedicated device.

The main benefit of resource dedication is that the kernel completely avoids any concurrent access to a resource from a different core. Resource dedication avoids the need for the kernel to implement critical sections to ensure mutual exclusion. In a non-resource-dedicated system with several cores, the overhead of dealing with multiple critical sections is a significant factor in limiting scalability.

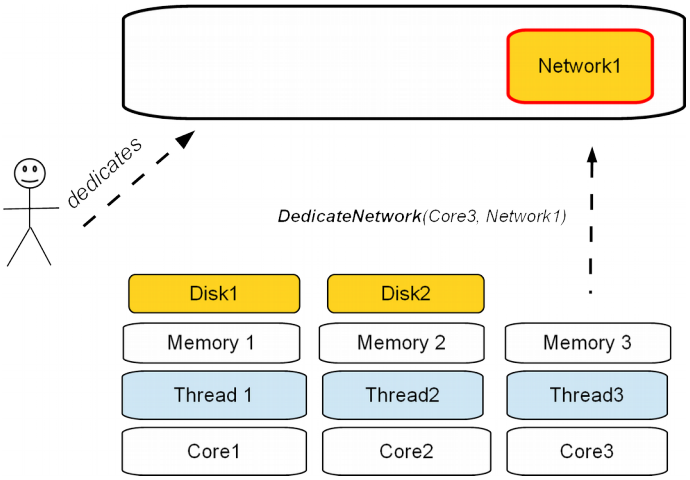

Figure 2. Dedication of Resources by Core.

Toro's architecture is designed on the basis of a cooperative scheduler and use of dedicated resources. Figure 3 illustrates the current state of the project. Currently, Toro supports up to 512 GB and is able to run on a x86-64 architecture. The file system has an Ext2 driver, and I am currently working on a FAT driver. The current file system drivers have been developed for IDE disks, and I am currently working on a driver for VirtIO block devices. The network stack supports e1000, ne2000 and VirtIO network cards.

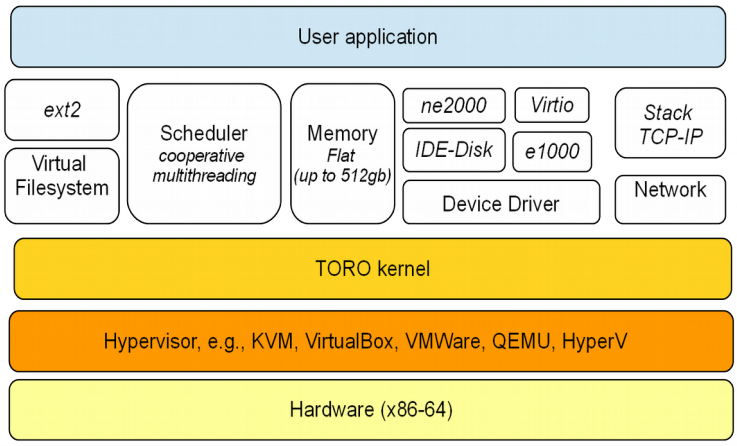

Figure 3. Current state of the Project.

The user application executes on top of the kernel. The following section explains how a Toro application must be structured.

Running an application in Toro

This section explains how to specify an application in Toro, based on the example application named ToroHelloWorldMicroService.pas. You can find the code for this example at https://github.com/MatiasVara/torokernel/blob/master/examples/ToroHelloWorldMicroservice.pas.
This application is a simple example of a micro-service that listens on port 80 and replies to each http request with a “Hello World”. This section is not meant to be a tutorial. Rather, it gives you an overview of the different sections needed in an application for Toro, explaining the example application via its source code.

```
.  program ToroHelloWorldMicroservice;
23 |
.  uses
25   Kernel in '..\rtl\Kernel.pas',
.    Process in '..\rtl\Process.pas',
.    Memory in '..\rtl\Memory.pas',
.    Debug in '..\rtl\Debug.pas',
.    Arch in '..\rtl\Arch.pas',
30   Filesystem in '..\rtl\Filesystem.pas',
.    Pci in '..\rtl\Drivers\Pci.pas',
.    Console in '..\rtl\Drivers\Console.pas',
.    Network in '..\rtl\Network.pas',
.    VirtIONet in '..\rtl\Drivers\VirtIONet.pas';
35
```

Figure 4. The opening lines of the example application.

The example application begins (as any Pascal application must begin) with the keyword **program**. After this, different units need to be imported (see lines 25 to 35 in Figure 4). The kernel functionalities are provided by five units: Process.pas, Memory.pas, Arch.pas, Filesystem.pas and Network.pas. Process.pas enables the user to create threads and implements the scheduler. Memory.pas enables the user to allocate and release memory. Filesystem.pas contains the entire file reading and writing API.
Once the kernel units are defined, the drivers need to be included. Our example uses networking, and in particular we are going to use a virtio network card, so the VirtIONet.pas unit is included (see line 34 in Figure 4). The network card driver could be different from this. For example, we might want to include a driver for the Intel gigabit (i.e. e1000.pas), or a driver for the ne2000 network card (i. e. ne2000.pas). In any case, the developer must use only the correct driver appropriate to his hardware.

Figure 5. User code initialization.

```
function PrintHelloWorld(entry: pchar): pchar;
begin
  Result := HelloWorldMsg;
end;

begin
  // dedicate the virtio network card to local cpu
  DedicateNetwork('virtionet', LocalIP, Gateway, MaskIP, nil);

  // register service callback
  ServiceHandler.DoInit    := @ServiceInit;
  ServiceHandler.DoAccept  := @ServiceAccept;
  ServiceHandler.DoTimeOut := @ServiceTimeOut;
  ServiceHandler.DoReceive := @ServiceReceive;
  ServiceHandler.DoClose   := @ServiceClose;

  // register the main service function
  MyMicroFunction := @PrintHelloWorld;

  // create the service
  SysRegisterNetworkService(@ServiceHandler);

  WriteConsoleF('\t /VToroService/n: listening on port %d ...\n',[SERVICE_PORT]);

  SysSuspendThread(0);
end.
```

The execution of the user application starts at line 235 (see Figure 5). At this point, all kernel and driver units have been initialized so the user is able to allocate and release memory or create threads. Any devices needed must be dedicated to a core by the user before first use. In our micro-service example, the application dedicates the network card to the current core at line 235 of Figure 5. This is done using the ***DedicateNetwork()*** routine. Next, the user application has to register the service that listens on port 80. To do this, it registers a set of callbacks using the function ***SysRegisterNetworkService()*** at line 248 of Figure 5. At this point, the kernel will handle all the tcp/ip requests and will invoke the callbacks when needed.

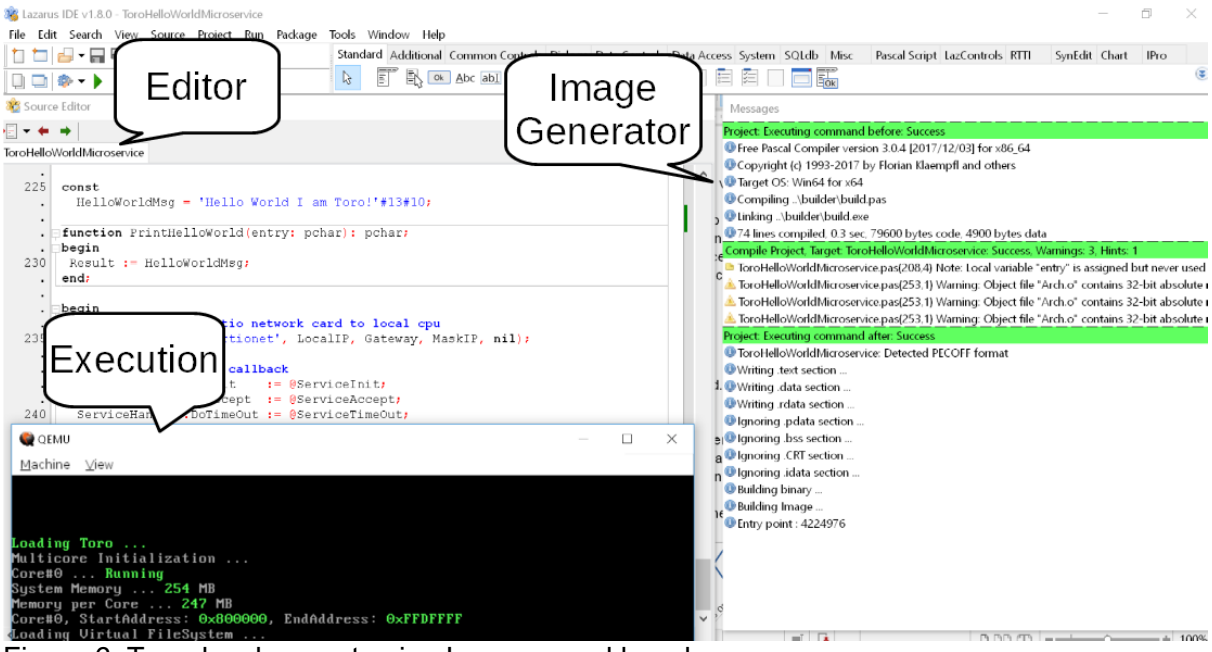

Figure 6. Toro development using Lazarus workbench.

I use Lazarus to develop applications in Toro. From the Lazarus IDE, I am able to automate the building of the kernel and its execution over QEMU. For example, if I open the project **ToroHelloWorldMicroservice.lpi** and I do **Compile**, the image is generated (see Figure 6). Then, I click on **Run** and a QEMU instance is created. The QEMU instance executes the image that contains the kernel and the application (see Figure 6).
Figure 7 illustrates how the image is built. The compilation results in an image that can be burned to a usb-key to run on bare metal, or used to create a Virtual Machine to run it on top of a hypervisor. You can use the same image in QEMU, VirtualBox, KVM, HyperV or Xen.

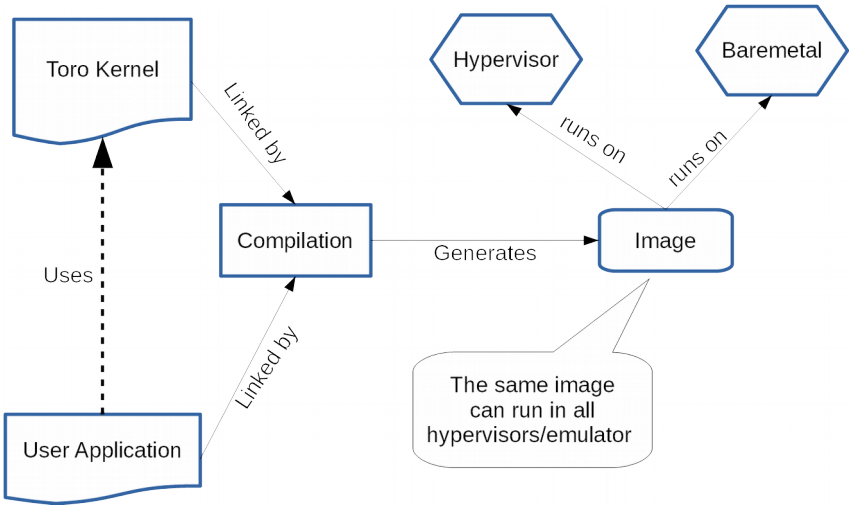

Figure 7. Compilation of the user application in Toro.

Conclusion

This article has presented Toro, a kernel optimized for running a single dedicated application. Toro has been written using Lazarus and the Free Pascal compiler, and is able to run either on bare metal or on top of a hypervisor. To write an application in Toro, a developer needs to understand both the underlying architecture and also the Toro API. An application running in Toro is able to use all system resources and avoids any interference from an intermediate Operating System. The project is still in a beta phase and the remaining work falls under several different categories, some of which are mentioned below.
One interesting use case for Toro is in the development of microservices. Such microservices run alone in a Toro guest on top of a hypervisor. Improvements in its TCP-IP Stack mean that microservices can support multiple concurrent connections with a lean memory footprint and efficient CPU usage.
Another interesting problem that I am tackling is the reduction of the CPU consumption of a Toro guest. This is a very important issue in production because the VCPU is a resource that is shared among all the VMs. I presented part of this work at FOSDEM'18.
Concerned to offer more drivers for Toro, I started to develop VirtIO drivers. For example, I just finished the networking driver and I am going to start the block driver soon. This will enable Toro to access devices at nearly bare metal speed when it runs as a KVM guest.
I am very concerned about the development of applications in Toro. I know how important it is to make coding, testing and debugging an application as straightforward and simple as possible. For this reason I plan to integrate Lazarus with QEMU and GDB. The idea is to be able to execute the kernel stepwise from within the IDE.
To help organize this ongoing work, I rely on issues users raise via the GitHub wiki. I use epics for new features and work items for shorter tasks. Some of these items are ideal for starting to explore Toro, so feel free to pick one up. For further details about how to contribute to Toro development, please visit the wiki in GitHub.

Acknowledgements

I want to thank the editors of this lovely magazine for this opportunity to write an article about Toro, which I have really enjoyed. I would like to thank my friend Cesar who has helped me in the review process. I would like to thank my Argentinian family for all their support.

Matias E. Vara Larsen
www.torokernel.io
github.com/MatiasVara/torokernel
matiasevara@gmail.com