

An overview of the XtratuM nanokernel *

M. Masmano, I. Ripoll, and A. Crespo

Universidad Politécnica de Valencia, Spain.

{mmasmano, iripoll, alfons}@disca.upv.es

Abstract

This paper presents a new nanokernel (XtratuM) which is aimed for executing several operating systems (where, at least, one of them is a real-time operating system) in the same hardware with temporal and spatial isolation.

Simplicity is the main idea behind of its design, therefore XtratuM can be defined as a thin layer of software which abstract the essential devices to run a kernel: the memory, the timers and the interrupts.

Besides, this paper presents the ARINC specification 653-1, an interface specification which allows to build high-reliability applications, being this interface a solid candidate to be the future XtratuM interface.

Keywords: Nanokernel, real-time, embedded system

1 Introduction

The word XtratuM derives from the latin term *substratum* referring to the layer of material which gives support to the upper layers.

In computer science, this term is used to designate a software hiding the programming complexity of the lower levels and supplying a functionality to the upper levels. From this point of view, XtratuM can be defined as a layer which is directly inserted between the hardware and others Operating Systems (OSes), easing the programming of these OSes as well as allowing to execute all of them in an isolated and concurrent way.

An Operating System (OS) can be defined as an abstraction layer between the physical hardware and the applications [6, 8, 7], which hides the underlying hardware

(processor, physical memory, storage mediums, etc) and provides a high-level abstraction of the machine instead (file system, process, threads, etc).

The use of an OS provides a big quantity of benefits to the applications mainly because the complexity of the hardware is hidden, simplifying the design and increasing the portability of the applications. Before the apparition of the OSes, the applications themselves were the responsible of setting up and managing the underlying hardware. Hence, these applications used to be designed for a machine with an specific configuration (quantity of memory, storage space, speed, etc), and therefore stopping working when this configuration was changed.

The design of an OS tend to be mainly guided by the requirements of the applications that will be run on it. It means that each OS just enables the use of a concrete range of applications, whereas it is not optimum or even it is useless when is used by applications with different requirements. For instance, the Windows OS was designed to supply a powerful, intuitive graphic interface, easy to use for the beginners. Windows is a clear example of an OS which does not provide any hard real-time capability (it is important to note that Windows offers the possibility of using an scheduler based on fixed priorities, but it does not turn windows into a hard real-time operating system).

Another example is the OSes used in the mobile phones, these kind of OSes are designed to be executed on embedded systems with low resources (battery, processing power, memory and storage space). Often these systems tend to implement a poor graphic environment. Therefore heavy graphic applications can not be executed on mobile phones' OSes. A last example could be the real-time OSes whose timing behaviour is deterministic. These systems are outstanding to execute applications with timing requirements. However, these OSes use to

*This work has been supported by the European Commission project number IST-2001-35102 (OCERA).

lack in suppling a friendly user interface.

Current processors are more and more powerful allowing to execute more and more complex applications. These complex applications usually can be divided in several parts with different requirements. An example of this can be a engine control program with a graphic monitor. The application can be split in two parts: the control algorithm which interacts with the engine, with hard real-time and fault-tolerance requirements (where using a hard RTOS is compulsory) and the monitor, with graphical requirements (where it would be nice to have a desktop OS with graphic capabilities like Windows or Linux).

Enhancing a general purpose OS with real-time capabilities has already been tried with disappointing results. The most satisfactory results have been obtained by the RTLinux approach, adding a software layer (a hard RTOS) beneath a general purpose OS (Linux or FreeBSD). This software layer virtualises interrupts and executes the general purpose OS in the background as the lowest priority task of the system.

This approach (running a hard real-time OS jointly with general purpose OS within the same machine) shows that running several OSES on the same machine enhances the execution of the applications with disperse requirements.

Some existing techniques to execute several OSES in the same machine are described below:

1. Running several OSES (guest OSES) on the top of another OS (host OS) through a virtualisation program: This technique consist in, on the top of a host OS, creating a complete virtual hardware machine. Thus, allowing the execution of the guest OS even if it were compiled for a different physical hardware architecture. The advantage of this kind of approach is that the guest OS can be directly executed (no modifications to the OS are required). Nevertheless, this technique also presents an important drawback: the guest OS can not be directly executed by the real processor but it has to be interpreted by the virtualisation program with a lost of performances. Therefore this approach is not useful to satisfy complex application requirements. Examples of virtual machines are: VMWare [9], Plex86 [3], win4lin [5].
2. Multiplexing the physical hardware between several OSES (guest OSES): Usually, OSES are internally

structured as a set of building blocks (see figure 1). These blocks can be divided in two categories, the device drivers: network-card drivers, PCI bus drivers, SCSI drivers, USB drivers, etc. And the abstractions of the hardware: memory manager, virtual filesystem infrastructure, network stack, etc. This method is implemented inserting a software layer beneath of the guest OSES. This software layer¹ takes over the real hardware and provides a virtual one to the guest OSES. An important disadvantage of this approach is that the current hardware architectures can not be completely virtualised, therefore the device drivers of the guest OSES have to be hacked to deal with virtual drivers rather real ones. Examples of implementations of this software layer are: Fiasco [4], Adeos [10], RTLinux [11], etc.

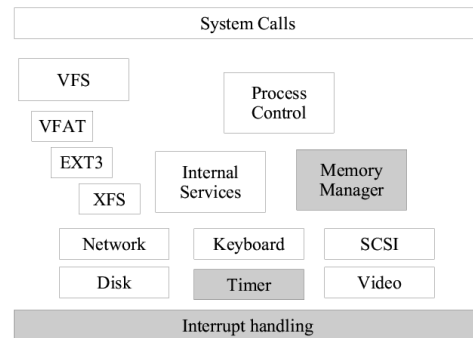


Figure 1: Classical operating system internal structure (simplified).

The results achieved using the second approach have encouraged us to design a new nanokernel, called Xtra-tuM, which has been explicitly designed to support the execution of at least one real-time OSES jointly one or more general OSES.

2 The ARINC specification 653-x

The ARINC specification 653-x defines a general-purpose APEX (APplication/EXecutive) between the OS of an

¹There are several ways of call this layer depending on the way it virtualise the hardware, its size and other implementation details. This denomination can be nanokernel, picokernel, and exokernel[2].

of hardware timers. For example, when XtratuM is executed in the intel x86 architecture (supposing the APIC timer available) will offer two different virtual timers: the classic PIT and the APIC timer. Besides, to work with these virtual timers, XtratuM also provides a high-level API to deal with them.

- Virtual Memory: Currently, XtratuM is only able to create a memory map per OS, enabling memory isolation among different OSes. This facility is still under development and lacks of many features like a sharing memory mechanism or increasing/shrinking the initial allocated memory.

Therefore, currently, from the guest OSes point of view, XtratuM just provides a high-level API to handle a timer and the interrupts. Each guest OS has to be aware about the rest of the existing hardware and how to share it. For instance, two different guest OSes which are going to use the serial port at the same time have to cooperate between them to avoid a race condition on it use.

New features like an inter-OS communication mechanism and a sharing resource protocol is being implemented and will be released soon.

3.1 XtratuM's scheduling issues

In its first releases, XtratuM implements an scheduling policy based on fixed priorities, where each domain has to indicate its priority at the creation moment. The reason of this decision is because, initially, XtratuM was though to execute a general purpose OS jointly with a hard RTOS in the same computer. In this conditions, triggered interrupts will be reissued to the domain depending on the domain's priority. The main benefits achieve because of the use of this policy is the low overhead and its implementation simplicity.

Nonetheless, the existence of more than one domain with timing requirements would make more desirable the use of a different scheduling policy. At this moment we are studying the possibility of introducing policies which guarantee the use of the CPU to each existing domain with timing constraints.

4 Implementation details

Implementing a nanokernel from the scratch is an arduous, hard task with a great amount of work to be carried out: programming a booting code for the targeted architecture, implementing new drivers, etc.

Nonetheless, as demonstrated in the Adeos nanokernel paper[10], all this work can be greatly simplified.

The nanokernel can be designed/implemented avoiding to start from the scratch but from a previously existing kernel (considered in Adeos as the root domain). The nanokernel is built around the infrastructure of this existing OS (the root domain). Using this approach, Adeos avoids the management of the virtual memory or the great majority of the existing devices, it just take care of interrupts of the system and supplies an scheduler to schedule the domains (guest OSes). Even the loading of the domains, as well as Adeos itself is implemented via the modules of the Linux kernel, therefore overriding the necessity of implementing a loader.

Taken advantage of this approach, the first versions of XtratuM have been built using the infrastructure supplied by the Linux kernel. Basically, these first versions consist of:

1. A patch for the Linux kernel. This patch modifies the Linux kernel in two ways: replacing all the disabling/enabling interrupt instructions by calls to XtratuM and inserting several hooks in the Linux kernel code. These hooks will be used later to virtualise the interrupts and the hardware timers.
2. The XtratuM nanokernel itself. Provided as a piece of software which has to be inserted inside Linux through the Linux kernel module mechanism. A XtratuM boot loader has been avoided using the Adeos approach, that is, XtratuM is loaded into the system as a Linux kernel module (sharing the memory map with Linux). However, XtratuM differs on the method used to load the guest OS. In Adeos the domains are also loaded as Linux kernel modules. XtratuM uses its own loader to create a specific memory map for each guest OS, enabling memory protection between the different OSes.

Besides, XtratuM virtualises the interrupts in a simi-

lar way as Adeos does. The IDT² entries, which contain the Linux's interrupt handlers, are replaced with the XtratuM's interrupt handler addresses. Once the IDT has been modified, interrupts are completely managed by XtratuM.

5 Examples of use

Next are some examples of how XtratuM can be used:

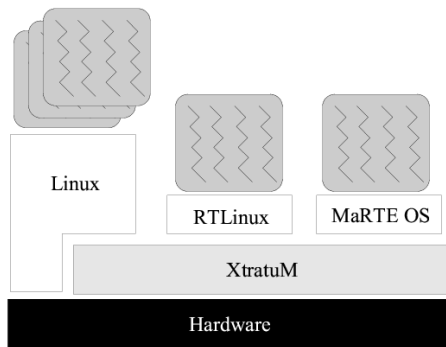


Figure 3: XtratuM running redundant systems.

Figure 3 represents a system where Linux is the background operating system; RTLinux/GPL playing the role of master real-time operating system and running the controlling application; and MaRTE OS also running the same application (but coded by a different developers group and using a different programming language Ada) but the application does not effectively send the actions to the hardware but compares its our results with those generated by the RTLinux/GPL domain. In case of a mismatch in the actions computed by both applications, or if a domain raises an exception, XtratuM can stop the buggy domain. Note that in order to know which is the faulting domain it might be possible to need a third domain.

XtratuM, when compiled with booting code (which will be developed soon) can be used to run the real-time operating system in several hardware processors with minor code changes. We are currently working on the ARM (Xscale) porting of XtratuM.

²Interrupt descriptors table, in the x86 architecture is the place where the address of the interrupt handlers are stored.

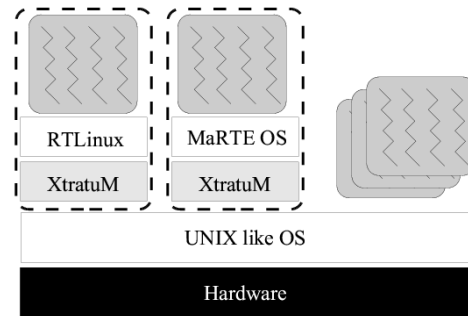


Figure 4: Using XtratuM to test the operating system, or application.

Another practical use of XtratuM framework is to run your RTOS on the top of a Linux system as a regular Linux process. The idea is to compile the guest operating system as a normal ELF executable and then run it the same way as ls or bash does. In this scenario, XtratuM used the POSIX signals and timers facilities provided by the host operating system as if they were interrupts and timers devices.

It is a very restricted and unrealistic system that can only be used for testing and to speed up the code development. It can also be used for teaching purposes.

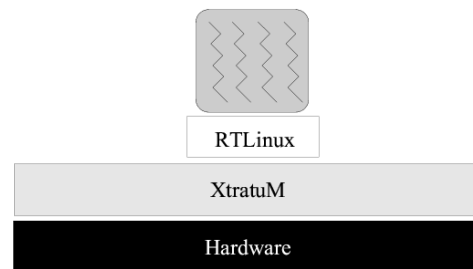


Figure 5: XtratuM as an Stand-alone system.

6 Conclusions

This paper presents the basic architecture of a new nanokernel called XtratuM which allows to execute several applications (application or operating system plus applica-

tions) with temporal and spatial isolation.

Currently, a prototype is ready where the Linux kernel is a domain executed in the XtratuM's memory space.

Although XtratuM was initially designed to permit the execution of a real-time operating system (as RTLinux/GPL) jointly with a general-purpose Operating System (Linux), currently, other configurations have been explored:

- Use of MaRTE OS replacing RTLinux/GPL as the real-time operating system.
- Execution of several real-time operating systems jointly with Linux. The approach requires a modification in the current XtratuM scheduling policy.
- Directly execution of an application without any kernel beneath it. For instance, a real-time application using a cyclic executive.

As future work, we are considering and working on three improvements:

- Adding new scheduling policies to XtratuM. These new policies will support more than only one application with timing requirements.
- Building an stand-alone version of XtratuM. Linux has to be executed as any other partition, with its own memory address. In the current version of XtratuM a fault in the Linux kernel is translated to a crash of the whole system.
- Replacing current non-standard XtratuM's API with a standard specification as ARINC 653-1.

References

- [1] L4 microkernels specification. <http://l4ka.org/projects/pistachio/l4-x2-r2.pdf>.
- [2] Mit exokernel operating system. <http://www.pdos.csail.mit.edu/exo.html>.
- [3] The new plex86 x86 virtual machine project. <http://plex86.sourceforge.net/>.
- [4] Herman Hartig et al. Fiasco microkernel. <http://os.inf.tu-dresden.de/fiasco/overview.html>.
- [5] NeTraverse. Win4lin. <https://www.netraverse.com>.
- [6] A. Silberschatz and P. B. Galvin. *Operating Systems*. Addison Wesley Longman, 1999.
- [7] A. S. Tanenbaum. *Modern Operating Systems. Second Edition*. Prentice Hall, 2001.
- [8] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems. Design and Implementation. Second Edition*. Prentice Hall, 2000.
- [9] Inc. VMWare. VMware workstation. <http://www.vmware.com/>.
- [10] Karim Yaghmour. Adaptive domain environment for operating systems. <http://www.opersys.com/ftp/pub/Adeos/adeos.pdf>.
- [11] Victor Yodaiken. The rlinux manifesto.