Operating System Design and Implementation

Libre Software Meeting 2005

In 2005, the Libre Software Meeting will take place in Dijon from July, 5th to July, 9th. Dijon is located to the East of France, 1h30 of train away from Paris. More information on the conference location and on accomodations can be found at http://www.libresoftwaremeeting.org

This document contains the program for the *Operating system design and implementation* topic of the Libre Software Meeting 2005. Many other technical or non-technical topics will be present at LSM 2005, their program are available on the official website.

For more information about the *Operating system design and implementation topic*, you can contact the topic chairmens Ludovic Courtès <ludovic *dot* courtes *at* laas *dot* fr> and Thomas Petazzoni <thomas *dot* petazzoni *at* enix *dot* org>.

Talks program

Mardi

Début	*Fin*	*Titre*	*Orateur(s)*	*Durée*
16h00	16h10	**Topic opening**	*Ludovic Courtès, Thomas Petazzoni*	10 min
16h10	17h40	**GNU Hurd, general presentation**	*Gaël Le Mignot*	1h30

Wednesday

Start	*End*	*Title*	*Speaker(s)*	*Duration*
9h30	10h40	**Porting the GNU Hurd to the L4 microkernel**	*Marcus Brinkmann*	1h10
10h40	11h00	**Break**		20 min
11h00	11h50	**A Market Based Approach to Resource Management for the GNU Hurd Multiserver Operating System**	*Neal Walfield*	50 min
11h50	12h40	**Measuring the Impact of the Virtual Memory Manager in Linux**	*Mel Gorman*	50 min
12h40	14h00	**Lunch**		1h20
14h00	15h10	**Plan 9 from Bell Labs**	*Charles Forsyth*	1h10
15h10	15h40	**A Look at the EROS Operating System** ***part 1 of the talk***	*Jonathan S. Shapiro* (Johns Hopkins University)	30 min
15h40	16h00	**Break**		20 min

Start	*End*	*Title*	*Speaker(s)*	*Duration*
16h00	16h30	**A Look at the EROS Operating System** ***part 2 of the talk***	*Jonathan S. Shapiro* (Johns Hopkins University)	30 min
16h30	17h40	**From EROS to Coyotos/BitC: Open Source meets Open Proofs**	*Jonathan S. Shapiro* (Johns Hopkins University)	1h10

Thursday

Start	*End*	*Title*	*Speaker(s)*	*Duration*
9h00	9h50	**Cluster Single System Image Systems: a State of the Art**	*Christine Morin (IRISA)*	50 min
9h50	10h40	**Kerrighed: a Single System Image System for High Performance Computing on Linux Clusters**	*Renaud Lottiaux et Pascal Gallard (IRISA)*	50 min
10h40	11h00	**Break**		20 min
11h00	11h50	**openMosix**	*Moshe Bar*	50 min
11h50	12h40	***Scheduler Activations*** **: principles and implementation in the Linux kernel**	*Vincent Danjean*	50 min
12h40	14h00	**Lunch**		1h20
14h00	18h40	Plenary sessions		

Friday

Start	*End*	*Title*	*Speaker(s)*	*Duration*
9h00	9h50	**JNode.org; why Java is practical for modern operating systems**	*Ewout Prangsma*	50 min
9h50	10h40	**High-level yet policy free ?**	*Frode Vatvedt Fjeld*	50 min
10h40	11h00	**Break**		20 min
11h00	11h40	**Thoughts about hOp: Operating Systems without Newspeak**	*Jérémy Bobbio*	40 min
11h40	12h10	**SOS: a step-by-step implementation of a "do it yourself" Unix-like OS**	*David Decotigny and Thomas Petazzoni*	30 min
12h10	12h40	**Toy Lovelace: an adaptation of SOS in Ada 95**	*Xavier Grave*	30 min
12h40	14h00	**Lunch**		1h20
14h00	14h50	**Improving System Dependability Using Virtual Machines**	*Joshua LeVasseur*	50 min

Start	*End*	*Title*	*Speaker(s)*	*Duration*
14h50	15h40	**User-Mode-Linux (UML)**	*Jeff Dike*	50 min
15h40	16h00	**Break**		20 min
16h00	16h50	**THINK, a software framework for component-based operating system kernels**	*Juraj Polakovic*	50 min
16h50	17h40	**The Bossa Framework for Scheduler Development**	*Julia Lawall (DIKU, University of Copenhagen)*	50 min

Saturday

Start	*End*	*Title*	*Speaker(s)*	*Duration*
9h00	10h40	**Wonderful trip inside an OS internals for the dummies** *part 1, in classroom*	*Thomas Petazzoni*	1h40
10h40	11h00	**Break**		20 min
11h00	12h40	**Wonderful trip inside an OS internals for the dummies** *part 2, in classroom*	*Thomas Petazzoni*	1h40

Talks abstracts

GNU Hurd, general presentation, by Gaël Le Mignot

The GNU Hurd, set of servers running on top of a micro-kernel, is the core of the GNU project. This talk will explain its history, the reasons of its creation, its deep links with the GNU project and the GNU philosophy, and will concentrate on its architecture and its technical principles. More precisely, inter-process communication mechanism, translator system, Mach memory handling and token-based security model will be explained. To conclude the talk, a quick presentation of Hurd's future (L4) will be given, and could be an introduction to Marcus' talk.

Porting the GNU Hurd to the L4 microkernel, by Marcus Brinkmann

A peasant revolution in the operating system world

The GNU Hurd is a multi-server operating system running on top of a microkernel. Although its current implementation on the basis of Mach provides a reasonably working prototype, allowing to run hundreds of different applications written for POSIX systems, some fundamental flaws can be identified which compromise the efficiency and robustness of the system. Is the microkernel experiment dead?

While the Real World (TM) has continued to build its success on proven, old concepts of operating system design, stretching them to their outer limits, the marginalized operating system research has been on-going. New generations of microkernels, like the L4 microkernel developed by Jochen Liedtke in Karlsruhe, have been designed to address fundamental flaws in the design of microkernels of the first generation.

The GNU Hurd port to the L4 microkernel is an effort to try to put these new design principles into practice, while preserving or even enhancing the user freedom that lies at the core of the Hurd design.

In my talk I will explain the two core defects we have identified in the Mach's RPC system and memory management, and describe how our attempt to fix them looks in the context of the L4 microkernel Pistachio. I will illustrate how these solutions enhance the GNU Hurd and fulfill its design philosophy at an even deeper level. I will explain why we are hopeful that we can address the fundamental design flaws identified in Mach, but I will also point out new problems appearing on the horizon.

The microkernel experiment continues.

Scheduler Activations : principles and implementation in the Linux kernel, by Vincent Danjean

Multithreading is increasingly used in high-performance computing to make full use of SMP systems. However, POSIX thread libraries are rarely adapted : useless functionnalities (signal handling, ...) or missing (userspace scheduling management, ...), threads eating too much resources... Some ad-hoc userspace libraries have been developped, but a system call issued by any thread blocks the entire application. The mecanism of *Scheduler Activations* overcomes this limitation. The original mecanism proposed by Anderson will be exposed, along with the modification that have been made for the Linux kernel and more specifically for the high-performance computing domain. Some results obtained with this mecanism and the Marcel thread library will also be presented.

Plan 9 from Bell Labs, by Charles Forsyth

Plan 9 is a distributed operating system written afresh by the research centre at Bell Labs (the one that wrote Unix), released several years ago as Free Software. The system offers a lean implementation of some simple but powerful mechanisms for building distributed systems.

The talk (or talks), will provide an overview of Plan 9's components, and its design and implementation. The system represents all resources (including graphics, network interfaces, protocol stacks, and services), as files in a hierarchical name space, local to a process and assembled dynamically. Universal representation as files allows one simple file-service protocol (9P) to implement services and link things together. It provides strong structural support at the system level for building distributed applications, replacing a plethora of special-purpose protocols. Many services are therefore implemented as file servers: the window system (rio), a ``user interface for programmers'' (Acme), inter-process communication (*plumbing*), service discovery, the domain name server (dns), and secure authentication (factotum). The disc-based file server (fossil) also has unusual characteristics: it provides a naming structure above an underlying archiving service (venti), and allows direct access by name to the files of yesteryear.

Key applications, including most of those mentioned above, rely on Plan 9's clean support for concurrent programming. Similar principles are applied inside the kernel itself. For instance, a device driver serves a set of names, representing data and control paths to the underlying device (or network). The system is portable: it runs native on x86, PowerPC, ARM, SPARC and other architectures. It is also self-supporting: it provides its own compiler suites and commands.

Improving System Dependability Using Virtual Machines, by Joshua LeVasseur

I describe our technique to reuse unmodified device drivers and to improve system dependability using virtual machines. We run the unmodified device driver, with its original operating system, in a virtual machine. This approach enables extensive reuse of existing and unmodified drivers, independent of the OS or device vendor, and significantly reduces the barrier to building new OS endeavors. By confining distinct device drivers to separate virtual machines, this technique isolates faults caused by defective or malicious drivers, and thus improves a system's dependability.

We demonstrate the driver reuse with a Linux 2.6 and Linux 2.4 para-virtualization environment. We show that our technique requires minimal support infrastructure, provides good performance, and provides strong fault isolation. Our prototype's network performance is within 3--8% of a native Linux system. Each additional virtual machine increases the CPU utilization by about 0.12%. We have successfully reused a wide variety of unmodified Linux network, disk, and PCI device drivers.

Cluster Single System Image Systems: a State of the Art, by Christine Morin (IRISA)

Single System Image (SSI) systems for clusters have recently gained a lot of interest, in particular in the area of high performance computing. A single system image system provides the illusion that a cluster is a single machine. Such a system eases cluster use and programming for parallel computing. A SSI globally manages all the cluster resources to hide resource distribution in the cluster nodes. It is made up of a set of distributed services to manage processes, memory, data streams and files cluster-wide.

In this talk, we focus more specifically on single system image systems built as an extension of Linux operating system, such as OpenSSI, openMosix and Kerrighed. These systems provide the Linux interface extended with system calls dedicated to clusters such as process migration. These systems are of interest as they allow the execution of a range of existing applications targeted for Linux-based SMP machines without modification.

First, we present the design of single system image systems and related implementation issues. Then, we analyze the results of a comparative study of OpenSSI, openMosix and Kerrighed open source Linux-based SSI systems. This study comprises of two parts: SSI properties coverage and a performance evaluation of key mechanisms of SSI systems.

Kerrighed: a Single System Image System for High Performance Computing on Linux Clusters, by Renaud Lottiaux and Pascal Gallard (IRISA)

Kerrighed is a single system image system (SSI) for high performance computing on clusters. It has been designed and implemented in the PARIS INRIA project-team at IRISA in the framework of a collaboration with EDF R&D. Based on Linux which it extends, Kerrighed is an operating system which virtualizes all the cluster resources and deals with resource sharing in multi-programming. Kerrighed provides a virtual shared memory multiprocessor (SMP-like) under Linux.

The software architecture of Kerrighed comprises of three layers. At the lowest level, Kerrighed implements a high performance and high reactivity communication service providing a kernel level interface well-suited to the construction of the higher level system services. Kerrighed communication service is portable regarding networking technologies (GigabitEthernet, Myrinet, Infiniband, ...). The intermediate layer implements the key concepts of Kerrighed: containers for global memory management and data sharing, ghost process for global processus management and checkpointing, dynamic data streams for global stream management in presence of process migration. In the upper layer, Kerrighed offers different services such as a global scheduler, a distributed file system, a memory management service and the traditional inter-process communication interfaces (pipe, sockets, ...).

Kerrighed differs from other single system image systems due to its high degree of customization. Kerrighed allows in particular to hot-change the global process scheduling policy. Moreover, several SSI functionalities can be enabled or disabled on demand on a per process basis using a programming or a command line interface.

Kerrighed provides binary compatibility for MPI (validated on MPICH), Posix multithreaded and OpenMP (validated with two OpenMP compilers targeting Pthreads) applications. Experimentations with industrial applications have been carried out in the framework of a contract with DGA.

In the talk, we present Kerrighed design principles and implementation and discuss some experimental results.

JNode.org; why Java is practical for modern operating systems, by Ewout Prangsma

JNode.org, a new operating system for personal use on modern hardware, has made a significant progress over the last two years and has now become the only actively developed java operating system in the open source community.

The JNode.org operating system is fully implemented in the java programming language and as such contains a java virtual machine (implemented in java) right in the hart of the OS. This makes the traditional kernel design of kernel spaces and user spaces obsolete. A flexible plugin framework has been implemented to make the OS very modular. All drivers, network layers, filesystems are seperate plugins that can be loaded, unload and reloaded at will. A modern device framework that uses the possibilities of the Java language makes it relatively easy to design & implement drivers that can cope with an ever evolving environment.

In this talk, the basic architecture & design of this OS will be discussed, followed by a more detailed look at the plugin and driver framework. It will give insight it this new exiting OS and will answer the question why Java is practical for modern operating systems.

High-level yet policy free ? By Frode Vatvedt Fjeld

Movitz is a run-time environment and Common Lisp compiler that targets x86 hardware, and is intended to serve as a platform for operating system kernels and single-application (embedded) systems. An important aspect of Movitz is its policy-free design: the application or OS designer remains free to provide his own overall system architecture and implementations for such basic mechanisms as garbage collection, threading, protection, and dynamic binding. This talk presents an overview of the Movitz platform, and outlines some of the challenges we faced in the design and implementation of a programming environment that is both high-level and policy-free.

THINK, a software framework for component-based operating system kernels, by Juraj Polakovic

Strictly speaking, THINK is not an operating system but a software framework for component-based operating system kernels. In other words, THINK allows the construction of specific operating systems by assembling and specializing components while encouraging code re-use. Many system design choices are left to the system designer and are not hardwired within the THINK platform. For example, the THINK framework can be used to build a single-address-space OS without hardware memory protection as well as an OS with virtual address spaces. THINK focuses on embedded real-time operating systems, i.e. operating systems with tight resource constraints and deterministic performance. The component architecture of THINK leads to small-footprint OS kernels with all the unnecessary functionality and features removed. The THINK software framework comes bundled with a library of reusable component that provides various operating system services such as CPU scheduling, interrupt handling, networking and file systems. Most of these components are hardware-independent and hardware dependencies when they occur are isolated within well-identified sub-components. The THINK framework also allows the development of extensions to implement non functional properties such as QoS, security and dynamic reconfiguration. After a brief description of THINK, the presentation will focus on mechanisms allowing the implementation of such non functional extensions.

A Look at the EROS Operating System, by Jonathan S. Shapiro (Johns Hopkins University)

What would it take to build a defensible, robust operating system? More importantly, an operating system that could support defensible and robust application construction? Not just defensible services, but defensible in the sense that every user is defended? In short, the operating system we need in the 21st century (and perhaps, in hindsight, the 20th). The requirements list is surprisingly short, and there is no operating system today that even begins to meet it. The EROS system tries to address this need. The talk will proceed in three phases.

First, I'll briefly go through an everyday application, a web browser, explain why its vulnerability is structural, and (by way of illustration) show how to make it defensible by refactoring using capability-based structuring techniques. Along the way, I'll point out various features of the operating system interface that need to be disabled in order for this restructuring to be effective. By the end of this, it should be clear why UNIX, Windows, and similar kernel architectures aren't the right starting point for architecting applications this way. I'll close this section by enumerating the design principles that it illustrates.

Second, I'll briefly describe the services provided by the EROS kernel, and identify some places where the EROS design is radically different from current operating system kernels -- particularly its reliance on fully accountable resource allocation and the presence of the checkpointing subsystem. Time will not permit a high-level system overview; the purpose of this section is to illustrate that the EROS kernel and critical subsystems provide a nearly ideal substrate for building efficient, defensible applications.

Third, I'll talk about the technical weaknesses in the EROS system, and why (at least in my opinion) the EROS system ultimately failed to produce a usable and complete system. Most of the technical issues are architecturally minor and have straightforward solutions. One has far-reaching implications. The larger causes are broader and present an opportunity for the open source community -- surprisingly minor changes to our current toolkit APIs would enable us to carry our applications forward onto a more secure platform with only minor change, and it would be good to start the source code transition now.

The EROS project will be continuing independently under the name CapROS. If there is a message to take away from the EROS work, it is that it really is possible to build defensible systems, that it really cannot be done by extending legacy platforms. The subsequent session on the Coyotos system will discuss where we are going with the successor to the EROS work.

From EROS to Coyotos/BitC: Open Source meets Open Proofs, by Jonathan S. Shapiro (Johns Hopkins University)

Three successor systems are emerging from the EROS work: the CapROS project is carrying forward the EROS architecture as-is, the L4++ (L5?) project is creating a fully capability-based successor to L4, and the Coyotos system, which is our own successor. The Coyotos project has two objectives. The first is to address the technical shortcomings of EROS. The second is to reframe the way software customers think about security and reliability. We hope to start a move toward general systems code that is ``safe'' and critical systems code that is *openly* verified. Coyotos will be used both as a seed project to test our verification infrastructure and as an exemplar for how such things can be built. The Coyotos system will be built in two editions: a first version in C to validate the design followed by a second version in BitC, our verifiable systems programming language.

A driving concept in the Coyotos/BitC work is the notion of ``Open Proofs.'' Our view is that the Common Criteria process has entirely failed for two reasons. First, inspection and testing just aren't rigorous enough to produce defensible and robust systems (and cannot be). Second, the customer isn't paying for the validation. The vendor can therefore exploit price competition to effectively negotiate the quality of the validation downward (and is doing so). We propose that a better foundation would be open source implementations in which verification objectives are formally expressed and rigorously checked in a fashion that produces an openly available (and openly reproducible) proof trail. The customer (or a contractor working for them) can re-execute these proofs for themselves as a check on the vendor. We refer to this process and methodology as ``Open Proofs.'' In addition to providing a viable basis for testing security claims, open proofs coupled with open source enable customers to make localized changes and adaptations, after which they can determine whether they have inadvertently violated some key security or reliability property.

Over the last 25 years, some truly amazing progress has been made in the research community on program verification. The main impediment to wider adoption is that very few ``builders'' have really tried to engineer these technologies into mainstream-usable form. Today's program verification tools are pretty much ``all or nothing,'' and a lot of the things we need to express as systems programmers cannot be expressed or verified in such limited frameworks. There is a gap both in the tools and in the orientation of the tool creators. As hard-core system builders, my group's view is ``Let us express what we need to express, and see what subset of that we can use verification techniques to analyze. As we learn, we'll adapt the source code, the verification tools, and the verification objectives to do better and better. We need to do full verification on very few programs. For the rest, maybe we just want to have a better technique for checking our work.'' Using the BitC language and logical framework, the Coyotos core implementation will provide examples at a number of cost/benefit tradeoff points of how verification techniques can be used in systems code.

This talk will proceed in two phases. First, I'll describe the core services of the Coyotos kernel by way of a contrast to the earlier EROS work. Along the way I'll note which of these features have close relatives in the secure L4 successor. In phase two, I'll give some examples of properties we would like to verify in the Coyotos implementation, ranging from simple consistency checks that are fairly easy to express and think about to global correctness properties that are going to take a very great deal of work. Along the way, I'll explain why we believe that the more complicated properties may succeed in Coyotos where comparable properties have failed in previous systems.

User-Mode-Linux (UML), by Jeff Dike

The talk will be about current and future work on UML and how it relates to virtualization in general. My view on virtualization is that it will be a pervasive part of an OS and its applications rather than a separate, isolated package. I see UML as being an integral part of bringing pervasive virtualization to both the kernel and userspace since it is both a virtualized kernel, making it useful inside the host kernel, and a process, making it usable by processes. I will talk about the benefits that I see in making virtualization available in both these areas, how exactly UML will contribute to it, and the work that is ongoing.

Thoughts about hOp: Operating Systems without *Newspeak*, by Jérémy Bobbio

Nearly all operating systems have been written using the C language. While 35 years ago, C was the clear language of choice, we believe this has become a limitation in terms of design, code size or security. There has been continual research on programming languages during all that time, perhaps kernel hackers could also benefit from what has been found?

Based on the hOp/House experiment and the purely functional language Haskell, this talk will try to review some of the pros and cons of modern programming language features when applied to operating systems.

The Bossa Framework for Scheduler Development, by Julia Lawall (DIKU, University of Copenhagen)

Writing a new scheduler and integrating it into an existing OS is a daunting task, requiring the understanding of multiple low-level kernel mechanisms. Indeed, implementing a new scheduler is outside the expertise of application programmers, even though they are the ones who understand best the scheduling needs of their applications.

We propose a framework, Bossa, to allow application programmers to implement kernel schedulers easily and safely. This framework defines a scheduling interface that is instantiated in a version of the Linux kernel by an Linux expert using an approach based on a variant of Aspect-Oriented Programming. Schedulers are written using a domain-specific language (DSL) that provides high-level scheduling-specific abstractions to simplify the programming of scheduling policies. The use of a DSL both eases scheduler programming and enables verification that a scheduling policy is compatible with OS requirements. We have found that Bossa gives good performance in practice. In this talk, we present the Bossa DSL, its implementation in Linux 2.4, and its use in the context of multimedia applications.

[joint work with Gilles Muller, Ecole des Mines de Nantes-INRIA, LINA]

openMosix, general presentation, by Moshe Bar

openMosix is a Linux kernel extension for single-system image clustering. This kernel extension turns a network of ordinary computers into a supercomputer for Linux applications. Once you have installed openMosix, the nodes in the cluster start talking to one another and the cluster adapts itself to the workload. Processes originating from any one node, if that node is too busy compared to others, can migrate to any other node. openMosix continuously attempts to optimize the resource allocation.

We achieve this with a kernel patch for Linux, creating a reliable, fast and cost-efficient SSI clustering platform that is linearly scalable and adaptive. With openMosix' Auto Discovery, a new node can be added while the cluster is running and the cluster will automatically begin to use the new resources. There is no need to program applications specifically for openMosix. Since all openMosix extensions are inside the kernel, every Linux application automatically and transparently benefits from the distributed computing concept of openMosix. The cluster behaves much as does a Symmetric Multi-Processor, but this solution scales to well over a thousand nodes which can themselves be SMPs.

The openMosix Community is very active, contributing add-on applications and sharing helpful information with all users. The openMosix Add-Ons and Community page lists these shared applications. And, it is all GPL'd.

openMosix 2.6 has been rewritten from ground and made platform independent to the maximum. We also released support for PowerPC, AMD Opteron and EM64T. In this talk we'll go through the various components as they exist today: process migration, VMA movement, load balancing, shared memory, /proc interface.

We'll also see some usage scenarios that both emphasize the very best and very worst use cases for SSI.

Measuring the Impact of the Virtual Memory Manager in Linux, by Mel Gorman

A major concern of almost any operating system is the methods and mechanisms it uses to manage physical memory and, in many cases, how it is mapped to virtual memory. Hence, the Virtual Memory Manager (VMM) plays a critical role in the systems performance as almost all OS subsystems must interact with the VMM and in some cases, heavily interact with it.

Due to its critical nature, it is important to understand all aspects of how the VMM interacts with the operating system and the different types of processes that run on the machine. Our research strives to understand and define metrics measuring the performance of a VMM and implement meaningful tools specifically for Linux.

This talk will start by introducing some of the performance concerns of the VMM and how they might be measured. We will then introduce some of the tools and metrics we have developed.

SOS: a step-by-step implementation of a "do it yourself" Unix-like OS, by David Decotigny and Thomas Petazzoni

We introduce Sos, (yet another) *Simple Operating System*, an Operating System aiming at learning in a step-by-step and didactic way how OSes are implemented. Monthly, an article in the French *GNU/Linux Magazine* review presents a new concept, describes it, quickly deals with its implementation in various production-grade OSes and finally proposes a simple yet functional approach to implementing it. Each article comes with the associated code (in english), resulting in the incremental implementation of the whole OS. In the end, {\sc Sos} proves to have the majority of common features found in Unix-like OSes, such as kernel/user threads, processes, virtual memory management, file systems support and basic networking. Technically speaking, *Sos* is written in C and targets common monoprocessor *IA32* machines. It derives from Kos (*The Kid Operating System*) in many respects, especially for the experience gathered by both authors along *Kos*'s existence. As for *Kos*, the main ambition of *Sos* isn't to become a new "killer-OS", but simply to serve as a base for OS education or as a valuable source of inspiration to develop new Unix-like OSes, Toy Lovelace being our first "spin-off"!

Toy Lovelace: an adaptation of SOS in Ada 95, by Xavier Grave

Toy Lovelace is a project which aims at demonstrating that writing an OS in Ada 95 is possible, and have fun while creating it.

Based on an adaptation of SOS, Toy Lovelace should benefit from Ada 95 advantages : exception handling, transparent task handling, good code readibility, strong typing and object oriented programming.

The adaptation of SOS is progressing and so is the addition of parts of the Ada run-time (object-oriented programming, Ada exception handling, ...).

The presentation will include explanations concerning the adaptation, based on the following comparaisons :

- C code vs. Ada code
- C compiler vs. Ada compiler
- generic code vs. macros

The presentation will also discuss how it is possible to embed an Ada runtime in the kernel.

Wonderful trip inside OS internals, for the dummies, by Thomas Petazzoni

This presentation aims at presenting the internals of a Unix system, and particularly its kernel. It is specifically targeted to beginners in operating systems topic. The main subsystems of a classical monolithic kernel such as Linux will be studied in a pedagogic way. At the end of this presentation, participants should understand the main mechanisms of a kernel, like process management, scheduling, virtual memory management, file system and device drivers.

The presentation really aims at being easy to grasp. Given by one of the authors of the articles concerning the SOS operating system published in GNU/Linux Magazine France, the conference will allow everybody to ask their questions about operating system internals.

The presentation will take a rather unorthodox form: instead of being given in a lecture hall, we will be in a class room, for a more interactive and open session. A subscription sheet will be available to sign up for this "class" in the classroom dedicated to the OS topic.

The presentation will take place either in English or French, depending on the audience.