University of Glasgow | School of Computing Science

Dependable Kernels and Device Drivers

Advanced Operating Systems (M)
Tutorial 5

Tutorial Outline

- Review of lectures
- Key learning outcomes
- Discussion

Review of Lectures (1)

- Dependable device drivers
  - Sources of bugs in device drivers
    - Based on a survey of Linux device driver bugs: 23% general, 38% hardware device handling, *39% concurrency and interactions with other kernel subsystems*
  - Improving device drivers: engineering approaches
    - Applying object oriented languages and techniques
    - Example: Apple MacOS X I/O Kit
  - Advancing the driver model
    - Making device driver state machines explicit in the source code
    - Automatic verification of device driver code to ensure correct implementation of protocols for interaction with other kernel subsystems, and to ensure correct locking
    - Example: device drivers in Microsoft's Singularity operating system

Review of Lectures (2)

- Dependable Kernel Architectures
  - Moving on from the monolithic, C-based, kernel architecture
  - Memory models
    - Example: Java memory model
  - Safe languages
    - Well-typed languages don’t have undefined behaviour
    - Banishing the null pointer
    - Pattern matching and messages
    - Immutable data
    - Linear types
- Microkernels and strongly isolated systems
  - Traditional microkernel model with hardware memory protection
  - Microkernel with software isolated processes: use the type system to enforce isolation between software processes, with no need for hardware protection
  - Example: Singularity from Microsoft Research

Key Learning Outcomes

- Understand how managed code and advanced type systems might be used in the design and implementation of future operating systems
  - To help the implementation of subsystems, such as device drivers
  - To help structure the system architecture, in the form of software isolation and robust message passing and concurrency primitives

Discussion

- Does the model of a microkernel with software isolated processes look plausible to you?

- Advantages and disadvantages?

- Benefits and limitations?

Discussion

- The architectures we have considered rely on a core unsafe microkernel, around which the safe systems architecture is constructed

- To what extent can the microkernel be written in a safe language?
- Can some operations only be implemented in an unsafe manner?

Discussion

- Safe languages require more extensive runtime support and error checking → more overhead
  - E.g., checking array bounds, support for exception handling, garbage collection overheads
- To what extent do reducing context switch times and faster message passing compensate for this overhead in the software isolated process model?

Discussion

- Questions, comments, and discussion on the papers?



Dingo: Taming Device Drivers

Leonid Ryzhyk[1 2] Peter Chubb[1 2] Ihor Kuz[1 2] Gernot Heiser[1 2 3]
[1]NICTA [2]The University of New South Wales [3]Open Kernel Labs
Sydney, Australia
leonid.ryzhyk@nicta.com.au

Abstract

Device drivers are notorious for being a major source of failure in operating systems. In analysing a sample of real defects in Linux drivers, we found that a large proportion (39%) of bugs are due to two key shortcomings in the device-driver architecture enforced by current operating systems: poorly-defined communication protocols between drivers and the OS, which confuse developers and lead to protocol violations, and a multithreaded model of computation that leads to numerous race conditions and deadlocks.

We claim that a better device driver architecture can help reduce the occurrence of these faults, and present our Dingo framework as constructive proof. Dingo provides a formal, state-machine based, language for describing driver protocols, which avoids confusion and ambiguity, and helps driver writers implement correct behaviour. It also enforces an event-driven model of computation, which eliminates most concurrency-related faults. Our implementation of the Dingo architecture in Linux offers these improvements, while introducing negligible performance overhead. It allows Dingo and native Linux drivers to coexist, providing a gradual migration path to more reliable device drivers.

Categories and Subject Descriptors D.4.4 [*Operating systems*]: Input/Output; D.3.2 [*Language Classifications*]: Specialized application languages

General Terms Languages, Reliability, Verification

Keywords Concurrent Programming, Device Drivers, Domain-Specific Languages, Fault Avoidance, Reliability.

1. Introduction

While accounting for about 70% of OS code, drivers typically contain several times more errors per line of code than other system components [Chou 2001] and, according to recent studies, are responsible for up to 70% of system failures [Ganapathi 2006, Murphy 2004]. With the introduction of advanced hardware capabilities such as hot-plugging, power management, and vectored I/O, device drivers have increased in complexity and hence become even more error-prone.

This paper explores the factors that contribute to driver complexity and lead to buggy drivers. In analysing bugs found in real Linux drivers, we discover and demonstrate quantitatively that a large proportion of these factors result from the way drivers interface with the OS, and can be eliminated or mitigated by a better design of the driver-OS interface.

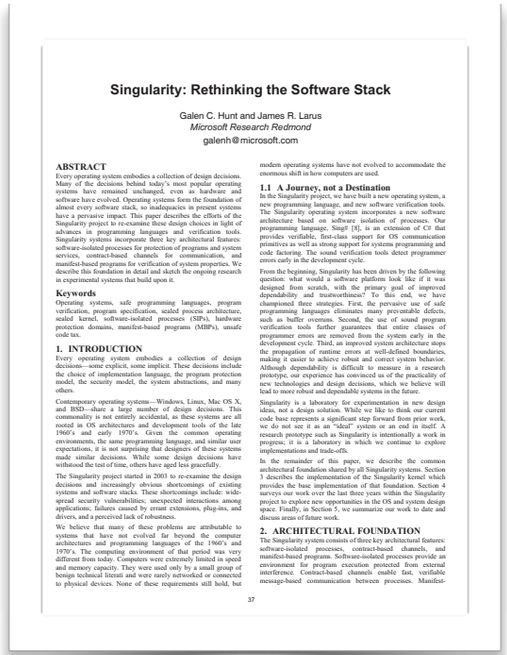
Specifically, we identify two shortcomings of the driver architecture common in modern operating systems: poorly-defined communication protocols between drivers and the OS, which confuse developers and lead to protocol violations, and a multithreaded model of computation that leads to numerous race conditions and deadlocks. To address these issues, we developed Dingo[1]—a device-driver architecture aimed at simplifying development and reducing the number of software defects in drivers.

In order to reduce protocol errors, driver protocols in Dingo are specified using a state-machine-based formal language called Tingu.[2] Tingu allows a clear and unambiguous description of requirements for driver behaviour, providing intuitive guidelines to driver programmers. The primary purpose of Tingu specifications is to serve as documentation helping driver developers avoid errors; however they can also be used as properties against which driver implementation can be formally validated either statically or at runtime. Presently we only support runtime validation by compiling driver protocol specifications into a runtime observer that detects protocol violations committed by the driver.

[1] A Dingo is an Australian wild dog.
[2] Tingu is an Australian aboriginal name for a Dingo cub.

[1] NICTA is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'09, April 1–3, 2009, Nuremberg, Germany.
Copyright © 2009 ACM 978-1-60558-482-9/09/04 ...$5.00



Singularity: Rethinking the Software Stack

Galen C. Hunt and James R. Larus
Microsoft Research Redmond
galenh@microsoft.com

ABSTRACT

Every operating system embodies a collection of design decisions. Many of the decisions behind today's most popular operating systems have remained unchanged, even as hardware and software have evolved. Operating systems form the foundation of almost every software stack, so inadequacies in present systems have a pervasive impact. This paper describes the efforts of the Singularity project to re-examine these design choices in light of advances in programming languages and verification tools. Singularity systems incorporate three key architectural features: software-isolated processes for protection of programs and system services, contract-based channels for communication, and manifest-based programs for verification of system properties. We describe this foundation in detail and sketch the ongoing research in experimental systems that build upon it.

Keywords

Operating systems, safe programming languages, program verification, program specification, sealed process architecture, sealed kernel, software-isolated processes (SIPs), hardware protection domains, manifest-based programs (MBPs), unsafe code tax.

1. INTRODUCTION

Every operating system embodies a collection of design decisions—some explicit, some implicit. These decisions include the choice of implementation language, the program protection model, the security model, the system abstractions, and many others.

Contemporary operating systems—Windows, Linux, Mac OS X, and BSD—share a large number of design decisions. This commonality is not entirely accidental, as these systems are all rooted in OS architectures and development tools of the late 1960's and early 1970's. Given the common operating environments, the same programming language, and similar user expectations, it is not surprising that designers of these systems made similar decisions. While some design decisions have withstood the test of time, others have aged less gracefully.

The Singularity project started in 2003 to re-examine the design decisions and increasingly obvious shortcomings of existing systems and software stacks. These shortcomings include: widespread security vulnerabilities; unexpected interactions among applications; failures caused by errant extensions, plug-ins, and drivers, and a perceived lack of robustness.

We believe that many of these problems are attributable to systems that have not evolved far beyond the computer architectures and programming languages of the 1960's and 1970's. The computing environment of that period was very different from today. Computers were extremely limited in speed and memory capacity. They were used only by a small group of benign technical literati and were rarely networked or connected to physical devices. None of these requirements still hold, but modern operating systems have not evolved to accommodate the enormous shift in how computers are used.

1.1 A Journey, not a Destination

In the Singularity project, we have built a new operating system, a new programming language, and new software verification tools. The Singularity operating system incorporates a new software architecture based on software isolation of processes. Our programming language, Sing# [8], is an extension of C# that provides verifiable, first-class support for OS communication primitives as well as strong support for systems programming and code factoring. The sound verification tools detect programmer errors early in the development cycle.

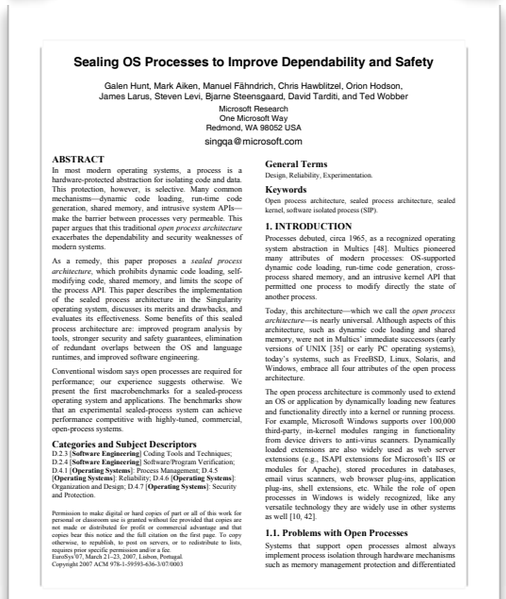
From the beginning, Singularity has been driven by the following question: what would a software platform look like if it was designed from scratch, with the primary goal of improved dependability and trustworthiness? To this end, we have championed three strategies. First, the pervasive use of safe programming languages eliminates many preventable defects, such as buffer overruns. Second, the use of sound program verification tools further guarantees that entire classes of programmer errors are removed from the system early in the development cycle. Third, an improved system architecture stops the propagation of runtime errors at well-defined boundaries, making it easier to achieve robust and correct system behavior. Although dependability is difficult to measure in a research prototype, our experience has convinced us of the practicality of new technologies and design decisions, which we believe will lead to more robust and dependable systems in the future.

Singularity is a laboratory for experimentation in new design ideas, not a design solution. While we like to think our current code base represents a significant step forward from prior work, we do not see it as an "ideal" system or an end in itself. A research prototype such as Singularity is intentionally a work in progress; it is a laboratory in which we continue to explore implementations and trade-offs.

In the remainder of this paper, we describe the common architectural foundation shared by all Singularity systems. Section 3 describes the implementation of the Singularity kernel which provides the base implementation of that foundation. Section 4 surveys our work over the last three years within the Singularity project to explore new opportunities in the OS and system design space. Finally, in Section 5, we summarize our work to date and discuss areas of future work.

2. ARCHITECTURAL FOUNDATION

The Singularity system consists of three key architectural features: software-isolated processes, contract-based channels, and manifest-based programs. Software-isolated processes provide an environment for program execution protected from external interference. Contract-based channels enable fast, verifiable message-based communication between processes. Manifest-



Sealing OS Processes to Improve Dependability and Safety

Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson,
James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber
Microsoft Research
One Microsoft Way
Redmond, WA 98052 USA
singga@microsoft.com

ABSTRACT

In most modern operating systems, a process is a hardware-protected abstraction for isolating code and data. This protection, however, is selective. Many common mechanisms—dynamic code loading, run-time code generation, shared memory, and intrusive system APIs—make the barrier between processes very permeable. This paper argues that this traditional *open process architecture* exacerbates the dependability and security weaknesses of modern systems.

As a remedy, this paper proposes a *sealed process architecture*, which prohibits dynamic code loading, self-modifying code, shared memory, and limits the scope of the process API. This paper describes the implementation of the sealed process architecture in the Singularity operating system, discusses its merits and drawbacks, and evaluates its effectiveness. Some benefits of this sealed process architecture are: improved program analysis by tools, stronger security and safety guarantees, elimination of redundant overlaps between the OS and language runtimes, and improved software engineering.

Conventional wisdom says open processes are required for performance; our experience suggests otherwise. We present the first macrobenchmarks for a sealed-process operating system and applications. The benchmarks show that an experimental sealed-process system can achieve performance competitive with highly-tuned, commercial, open-process systems.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**] Coding Tools and Techniques; D.2.4 [**Software Engineering**] Software/Program Verification; D.4.1 [**Operating Systems**]: Process Management; D.4.5 [**Operating Systems**] Reliability; D.4.6 [**Operating Systems**]: Organization and Design; D.4.7 [**Operating Systems**]: Security and Protection.

General Terms

Design, Reliability, Experimentation.

Keywords

Open process architecture, sealed process architecture, sealed kernel, software isolated process (SIP).

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'07, March 21–23, 2007, Lisbon, Portugal.
Copyright 2007 ACM 978-1-59593-636-3/07/0003

1. INTRODUCTION

Processes debuted, circa 1965, as a recognized operating system abstraction in Multics [48]. Multics pioneered many attributes of modern processes: OS-supported dynamic code loading, run-time code generation, cross-process shared memory, and an intrusive kernel API that permitted one process to modify directly the state of another process.

Today, this architecture—which we call the *open process architecture*—is nearly universal. Although aspects of this architecture, such as dynamic code loading and shared memory, were not in Multics' immediate successors (early versions of UNIX [35] or early PC operating systems), today's systems, such as FreeBSD, Linux, Solaris, and Windows, embrace all four attributes of the open process architecture.

The open process architecture is commonly used to extend an OS or application by dynamically loading new features and functionality directly into a kernel or running process. For example, Microsoft Windows supports over 100,000 third-party, in-kernel modules ranging in functionality from device drivers to anti-virus scanners. Dynamically loaded extensions are also widely used as web server extensions (e.g., ISAPI extensions for Microsoft's IIS or modules for Apache), stored procedures in databases, email virus scanners, web browser plug-ins, application plug-ins, shell extensions, etc. While the role of open processes in Windows is widely recognized, like any versatile technology they are widely use in other systems as well [10, 42].

1.1. Problems with Open Processes

Systems that support open processes almost always implement process isolation through hardware mechanisms such as memory management protection and differentiated

Summary

- Widely deployed operating systems are based on the heritage of monolithic, C-based, kernels
  - Linux is the newest system to obtain wide deployment, but has little innovation in terms of fundamental systems architecture
  - Research prototypes are beginning to explore radically different system architectures
- Exercise 3 now available