University of Glasgow

School of Computing Science

Systems Programming

Advanced Operating Systems (M)

Tutorial 1

Colin Perkins | https://csperkins.org/ | Copyright © 2018 | This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Goals of Tutorials

- Level M course → assessing critical thinking skills; ability to read research papers, extract key insights
- Tutorials intended to facilitate this:
  - To provide space to discuss the Further Reading highlighted at the end of the lectures in the previous week, to consolidate learning, and emphasise key points of the material
  - You are expected to have read the highlighted papers, and to come to the tutorial prepared to discuss the material
  - Write your own summaries of the papers: what are the key concepts and ideas? what isn't clear? what's unimportant detail?
  - Discuss material that isn't clear in the tutorials → you're not expected to understand everything in the papers

CC BY ND

Reading Research Papers

- Was everyone able to access the papers?
- Experiences with the process of reading a research paper?
  - Critical reading of a research paper is difficult and requires practice; read in a structured manner, not end-to-end, think about the material as you go
  - Advice on paper reading: http://www.eecs.harvard.edu/~michaelm/postscripts/ReadPaper.pdf
    - S. Keshav, “How to Read a Paper”, ACM Computer Communication Review, 37(3), July 2007 DOI: 10.1145/1273445.1273458
- Did everyone take notes? Come with questions to discuss?

CC BY ND

Discussion of Papers

- J. Shapiro, “Programming language challenges in systems codes: why systems programmers still use C, and what to do about it”, Proc. PLOS 2006, San Jose, CA, Oct. 2006. DOI:10.1145/1215995.1216004
  - Systems programming: constrained memory, I/O performance, data representation, state matters
  - Fallacies: factors of 1.5–2 don’t matter; boxed representation can be optimised; the optimiser can fix it; legacy issues insurmountable
  - Suggests: annotating code to check application constraints
  - Suggests: manual but automatically checked storage management; explicit control over data representation
  - The BitC project wasn’t a success, but are the ideas valid?



Programming Language Challenges in Systems Codes

Why Systems Programmers Still Use C, and What to Do About It

Jonathan Shapiro, Ph.D.
Systems Research Laboratory
Department of Computer Science
Johns Hopkins University
shap@cs.jhu.edu

Abstract

There have been major advances in programming languages over the last 20 years. Given this, it seems appropriate to ask why systems programmers continue to largely ignore these languages. What are the deficiencies in the eyes of the systems programmers? How have the efforts of the programming language community been misdirected (from their perspective)? What can/should the PL community do address this?

As someone whose research straddles these areas, I was asked to give a talk at this year’s PLOS workshop. What follows are my thoughts on this subject, which may or not represent those of other systems programmers.

1. Introduction

Modern programming languages such as ML [16] or Haskell [17] provide newer, stronger, and more expressive type systems than systems programming languages such as C [15, 13] or Ada [12]. Why have they been of so little interest to systems developers, and what can/should we do about it?

As the primary author of the EROS system [18] and its successor Coyotos [20], both of which are high-performance microkernels, it seems fair to characterize myself primarily as a hardcore systems programmer and security architect. However, there are skeletons in my closet. In the mid-1980s, my group at Bell Labs developed one of the first large commercial C++ applications — perhaps *the* first. My early involvement with C++ includes the first book on reusable C++ programming [21], which is either not well known or has been graciously disregarded by my colleagues.

In this audience I am tempted to plead for mercy on the grounds of youth and ignorance, but having been an active advocate of C++ for so long this entails a certain degree of *chutzpah*.[1] There is hope. Microkernel developers seem to have abandoned C++ in favor of C. The book is out of print in most countries, and no longer encourages deviant coding practices among susceptible young programmers.

A Word About BitC Brewer *et al.*’s cry that *Thirty Years is Long Enough* [6] resonates. It really *is* a bit disturbing that we are still trying to use a high-level assembly language created in the early 1970s for critical production code 35 years later. But Brewer’s lament begs the question: why has no viable replacement for C emerged from the programming languages community? In trying to answer this, my group at Johns Hopkins has started work on a new programming language: BitC. In talking about this work, we have encountered a curious blindness from the PL community.

We are often asked “Why are you building BitC?” The tacit assumption seems to be that if there is nothing fundamentally new in the language it isn’t interesting. The BitC goal isn’t to invent a new language or any new language concepts. It is to integrate existing concepts with advances in prover technology, and reify them in a language that allows us to build stateful low-level systems codes that we can reason about in varying measure using automated tools. The feeling seems to be that everything we are doing is straightforward (read: uninteresting). Would that it were so.

Systems programming — and BitC — are fundamentally about engineering rather than programming languages. In the 1980s, when compiler writers still struggled with inadequate machine resources, engineering considerations were respected criteria for language and compiler design, and a sense of “transparency” was still regarded as important.[2] By the time I left the PL community in 1990, respect for engineering and pragmatics was fast fading, and today it is all but gone. The concrete syntax of Standard ML [16] and Haskell [17] are every bit as bad as C++. It is a curious measure of the programming language community that nobody cares. In our pursuit of type theory and semantics,

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.
PLOS 2006, Oct. 22, 2006, San Jose, California, United States
Copyright © 2006 ACM 1-59593-577-0/10/2006…$5.00

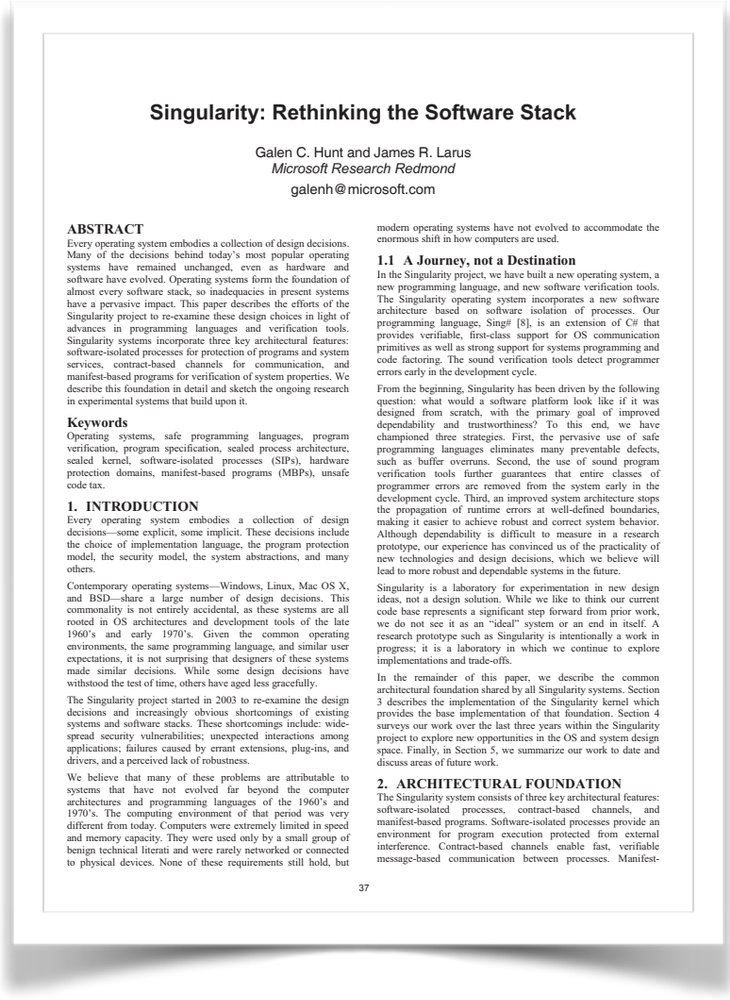
[1] *Chutzpah* is best defined by example. Chutzpah is when a person murders both of their parents and then asks the court for mercy on the grounds that they are an orphan.

[2] By “transparent,” I mean implementations in which the programmer has a relatively direct understanding of machine-level behavior.

CC BY ND

Discussion of Papers

- G. Hunt and J. Larus. “Singularity: Rethinking the software stack”, ACM SIGOPS OS Review, 41(2), April 2007. DOI:10.1145/1243418.1243424
  - Use of strongly-typed languages to build an operating system; software isolated processes; message passing – is this a sound basis for the system?
  - Type-safe message passing through channels; checked state machines for communication protocols (e.g., to control device driver state) – useful tool to help ensure correctness, or over-complex and stifling?
  - Small unsafe microkernel, with type-safe system layered above – can the microkernel be written in a safe language?
  - Threads and exchange heap; garbage collection –overheads?
  - Is the idea of running everything in a virtual machine reasonable?



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: wide-spread 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-

37

CC BY ND