University of Glasgow

School of Computing Science

# Operating Systems and Systems Programming

## Advanced Operating Systems (M)

## Lecture 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.

# Introduction and Course Administration

# Rationale

- Radical changes to computing landscape;
  - Desktop PC becoming irrelevant
  - Heterogeneous, multicore, mobile, and real-time systems – smart phones, tablets – now ubiquitous
  - Not reflected by corresponding change in operating system design practice
- This course will...
  - Review research on systems programming techniques and operating systems design;
  - Discuss the limitations of deployed systems; and
  - Show how the operating system infrastructure might evolve to address the challenges of supporting modern computing systems.

CC BY ND

# Aims and Objectives

- To explore programming language and operating system facilities essential to implement real-time, reactive, and embedded systems
- To discuss limitations of widely-used operating systems, introduce new design approaches to address challenges of security, robustness, and concurrency
- To give an understanding of practical engineering issues in real-time and concurrent systems; and suggest appropriate implementation techniques

CC BY ND

# Intended Learning Outcomes (1)

- At the end of this course, you should be able to:
  - Clearly differentiate the different issues that arise in designing real-time systems;
  - Analyse a variety of real-time scheduling techniques, and prove the correctness of the resulting schedule; implement basic scheduling algorithms;
  - Understand how to apply real-time scheduling theory to the design and implementation of a real-world system using the POSIX real-time extensions, and be able to demonstrate how to manage resource access in such a system;
  - Describe how embedded systems are constructed, and discuss the limitations and advantages of C as a systems programming language, understand how managed code and advanced type systems might be used in the design and implementation of future operating systems;

    …

CC BY ND

# Intended Learning Outcomes (2)

…

- Discuss the advantages and disadvantages of integrating garbage collection with the operating system/runtime, understand the operation of popular garbage collection algorithms, and alternative techniques for memory management, and know when it might be appropriate to apply such techniques and managed runtimes to real-time systems and/or operating systems;
- Understand the impact of heterogeneous multicore systems on operating systems, compare and evaluate different programming models for concurrent systems, their implementation, and their impact on operating systems;
- Construct and/or analyse simple programming to demonstrate understanding of novel techniques for memory management and/or concurrent programming, to understand the trade-offs and implementation decisions.

CC BY ND

# Course Outline

- Key topics in systems programming and operating systems:
  - Systems programming
  - Hardware trends
  - Memory management
  - Implications of concurrency
  - High performance networking
  - Virtualisation
- Focus is on ideas and advanced concepts
  - Research topics and new approaches

CC BY ND

# Systems Programming

- What do we mean by systems programming?
  - Systems programs interact with hardware
  - Systems programs have memory and data layout constraints
  - Systems programs strongly driven by bulk I/O performance
  - Systems programs maintain long-lived, concurrently accessed, state
  - This is writing operating system kernels, low-level services, embedded systems, etc.
- Why is systems programming challenging?
- How are systems programs implemented? How should they be implemented?

CC BY ND

# Hardware Trends

- How is hardware evolving? How does this affect systems programming?
  - Implications of Moore's law and Dennard scaling
  - Implications of concurrency
  - Implications of solid-state storage
  - Implications of high performance networking

CC BY ND

# Memory Management

- Implications of NUMA and the caching hierarchy
- Memory management
  - Layout of a processes address space
  - Manual memory management
  - Region based memory management; ownership and borrowing; Rust
  - Garbage collection algorithms

BY ND

# Implications of Concurrency

- Memory models, threads, and locks
- Why threads and shared state concurrency are not sufficient
- Alternative concurrency models
  - Transactions
  - Message passing: Akka, Erlang, etc.

CC BY ND

# High Performance Networking

- Interactions between high-performance networking and Moore's law
  - Why high-performance networking is becoming more challenging
- APIs and programming models
  - Message passing
  - netmap and StackMap
  - User-space protocol stacks

CC BY ND

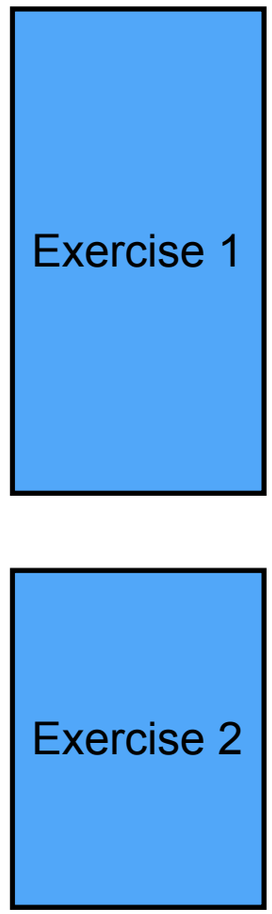
# Virtualisation

- What is virtualisation?
  - Hypervisors
  - Cloud computing
- Full system virtualisation
  - Xen
- Containers
  - FreeBSD jails, Docker, etc.
  - Unikernels

CC BY ND

# Timetable

| Week | Lecture | Subject |
|---|---|---|
| 1 | Lecture 1 | Operating Systems and Systems Programming |
| | | |
| 2 | Lecture 2 | Hardware Trends and their Implications |
| | Tutorial 1 | Systems Programming |
| 3 | Lecture 3 | Memory Management |
| | Tutorial 2 | Hardware Trends and their Implications |
| 4 | | No lectures or tutorial |
| | | |
| 5 | Lecture 4 | Garbage Collection |
| | Tutorial 3 | Memory Management |
| 6 | Lecture 5 | Concurrency |
| | Tutorial 4 | Garbage Collection |
| 7 | Lecture 6 | Message Passing |
| | Tutorial 5 | Concurrency |
| 8 | Lecture 7 | Network Programming Interfaces |
| | Tutorial 6 | Message Passing |
| 9 | Lecture 8 | Virtualisation |
| | Tutorial 7 | Networking |
| 10 | Tutorial 8 | Virtualisation |
| | Lecture 9 | Wrap-up |
| | | |




Colin Perkins | https://csperkins.org/ | Copyright © 2018

# 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

# Assessment

- Level M course; 10 credits
- Coursework (20%)

| Exercise | Weight | Topic | Set | Due |
|---|---|---|---|---|
| 1 | 12% | Memory Management | Lecture 3 | Tutorial 4 |
| 2 | 8% | Concurrency and Message Passing | Lecture 6 | Tutorial 7 |

- Examination (80%)
  - Two hours duration; sample and past papers are available on Moodle or the course website
  - All material in the lectures, tutorials, and cited papers is examinable
  - Aim is to test your understanding of the material, not to test your memory of all the details; explain why – don’t just recite what

CC BY ND

# Pre- and co-requisites

- Required pre-requisites:
  - Computer Systems 2
  - Operating Systems (H)
  - Advanced Programming (H)
  - Functional Programming (H)
- Recommended co-requisites:
  - Computer Architecture (H)

BY ND

# Required Reading

- No set text book – research papers will be cited:
  - DOIs are provided; resolve via http://dx.doi.org/ – all papers accessible for free from on campus (some may be paywalled from elsewhere)
  - You are expected to read the papers; it will be beneficial to follow-up on some of the references and do further background reading
    - Critical reading of a research paper is difficult and requires practice; read in a structured manner, not end-to-end, thinking about the material as you go; read claims carefully; realise that not everything written in a research paper is necessarily correct or a good idea – think and judge for yourself!
    - 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
- The contents of the highlighted “Further Reading” research papers is examinable – but exam questions covering that material will focus on concepts, rather than on details

BY ND

# Resources and Contact Details

- Lecture slides and other materials are on Moodle
  - Also https://csperkins.org/teaching/2017-2018/adv-os/
  - Printed lecture handouts will not be provided – learning is enhanced by taking your own notes during lectures and tutorials
- Course coordinator:
  - Dr Colin Perkins, Room S101b, Lilybank Gardens
  - Email: colin.perkins@glasgow.ac.uk
  - No assigned office hours – email to make appointment if needed

CC BY ND

# Systems Programming

# Systems Programming

- Why is systems-level programming different?
  - Systems programs interact with hardware
  - Systems programs have memory and data layout constraints
  - Systems programs strongly driven by bulk I/O performance
  - Systems programs maintain long-lived, concurrently accessed, state

- We are used to high-level programming, ignoring low-level details
  → the details matter when writing systems programs

CC BY ND

# Challenges

- Operating systems are evolving:
  - more power constrained
  - more real-time applications
  - more embedded
  - more concurrent
  - more safety critical
  - more security critical

- Are C and Unix the right programming model for the operating systems of the future?

CC BY ND

# Challenges: Power Constraints

- Many systems run on constrained hardware
  - May have limits on power consumption (e.g., battery powered)
  - May have to be physically small and/or robust
  - May have strict heat production limits
  - May have strict cost constraints
- Used to throwing hardware at a problem, writing inefficient – but easy to implement – software
  - Software engineering based around programmer productivity
  - Constraints differ in embedded systems – optimise for correctness, cost, then programmer productivity

CC BY ND

# Challenges: Ensuring Predictable Timing

- Real-time systems → scheduling theory can prove correctness, if system timing predictable
- Numerous sources of unpredictability
  - Timing variation due to dependence on algorithm input values → measure
  - Blocking due to resource access
  - Preemption by higher priority tasks or interrupt handlers
  - Processor cache improves average timing, with poor worst-case bounds
  - Virtual memory – address translation, paging, memory protection
  - Memory allocation and management – `malloc()` or garbage collector
- Avoid by defensive programming
  - Disable or avoid features that cause timing variation
  - Optimise for predictability, not raw performance

CC BY ND

# Challenges: Embedded Systems

- Constraints on embedded systems:
  - Must interact with hardware to manipulate their environment – custom device drivers and low-level hardware access in application code
  - Safety critical or simply hard to upgrade – strong correctness constraints
  - Often resource constrained, with a low-level programming model

- Issues differ from those inherent in traditional desktop application programming

CC BY ND

# Device Drivers

- Devices represented by bit fields at known address
  - Bit-level manipulation to access fields in control register
  - Code needs memory address and size of control register, layout, endianness, and meaning of bit fields within register
- C allows definition of bit fields and explicit access to particular memory addresses via pointers – needed for implementation of device drivers
- Illusion of portability – standard C does not specify:
  - Size of basic types (e.g., a `char` is not required to be 8 bits, an `int` is not required to be 32 bits, etc.)
  - Bit and byte ordering
  - Alignment or atomicity of memory access
  - Each environment defines these – e.g., `<stdint.h>` and `<limits.h>` – but type checking is limited
- Device drivers written in C a frequent source of bugs
- Other languages (e.g., Ada, Rust) provide strict definitions and allow stronger type checking

```
struct {
    short errors     : 4;
    short busy       : 1;
    short unit_sel   : 3
    short done       : 1;
    short irq_enable : 1
    short reserved   : 3
    short dev_func   : 2;
    short dev_enable : 1;
} ctrl_reg;

int enable_irq(void)
{
    ctrl_reg *r = 0x80000024;
    ctrl_reg  tmp;

    tmp = *r;
    if (tmp.busy == 0) {
        tmp.irq_enable = 1;
        *r = tmp;
        return 1;
    }
    return 0;
}
```

Example: hardware access in C

Colin Perkins | https://csperkins.org/ | Copyright © 2018

# Sources of Bugs in Device Drivers (1)

| Name | Description | Total faults | Device prot. violations | S/W protocol violations | Concurrency faults | Generic faults |
|---|---|---|---|---|---|---|
| USB drivers | | | | | | |
| rtl8150 | rtl8150 USB-to-Ethernet adapter | 16 | 3 | 2 | **7** | 4 |
| catc | el1210a USB-to-Ethernet adapter | 2 | **1** | 0 | **1** | 0 |
| kaweth | kl5kusb101 USB-to-Ethernet adapter | 15 | 1 | 2 | **8** | 4 |
| usb net | generic USB network driver | 45 | **16** | 9 | 6 | 14 |
| usb hub | USB hub | 67 | **27** | 16 | 13 | 11 |
| usb serial | USB-to-serial converter | 50 | 2 | 17 | 13 | **18** |
| usb storage | USB Mass Storage devices | 23 | 7 | 5 | **10** | 1 |
| IEEE 1394 drivers | | | | | | |
| eth1394 | generic ieee1394 Ethernet driver | 22 | **6** | **6** | 4 | **6** |
| sbp2 | sbp-2 transport protocol | 46 | **18** | 10 | 12 | 6 |
| PCI drivers | | | | | | |
| mthca | InfiniHost InfiniBand adapter | 123 | **52** | 22 | 11 | 38 |
| bnx2 | bnx2 network driver | 51 | **35** | 4 | 5 | 7 |
| i810 fb | i810 frame buffer device | 16 | 4 | **5** | 2 | **5** |
| cmipci | cmi8338 soundcard | 22 | **17** | 3 | 1 | 1 |
| | Total | 498 | 189 (38%) | 101 (20%) | 93 (19%) | 115 (23%) |

Fix through device documentation and better language support for low-level programming?

Can we address these through improvements to the supporting infrastructure for device-drivers?

Summary cause of bugs found in Linux USB, Firewire (IEEE 1394), and PCI drivers from 2002–2008
[from L. Ryzhyk *et al.*, “Dingo: Taming device drivers”, Proc. EuroSys 2009, DOI 10.1145/1519065.1519095]

Device protocol violations are mis-programming of the hardware, software protocol violations and concurrency faults are invalid interactions with the rest of the Linux kernel

CC BY ND

# Sources of Bugs in Device Drivers (2)

- What causes software protocol violations and concurrency faults?
  - Misunderstanding or misuse of the device driver API, especially in uncommon code paths (e.g., error handling, hot-plug, power management)
  - Incorrect use of locks leading to race conditions and deadlocks

| **Type of faults** | **#** |
|---|---|
| Ordering violations | |
| Driver configuration protocol violation | 16 |
| Data protocol violation | 9 |
| Resource ownership protocol violation | 8 |
| Power management protocol violation | 8 |
| Hot unplug protocol violation | 5 |
| Format violations | |
| Incorrect use of OS data structures | 29 |
| Passing an incorrect argument to an OS service | 19 |
| Returning invalid error code | 7 |

**Table 2.** Types of software protocol violations.

- Bad programming and poor documentation of kernel APIs and locking requirements?
- Or error-prone programming languages, concurrency models, and badly designed kernel APIs?

| **Type of faults** | **#** |
|---|---|
| Race or deadlock in configuration functions | 29 |
| Race or deadlock in the hot-unplug handler | 26 |
| Calling a blocking function in an atomic context | 21 |
| Race or deadlock in the data path | 7 |
| Race or deadlock in power management functions | 5 |
| Using uninitialised synchronisation primitive | 2 |
| Imbalanced locks | 2 |
| Calling an OS service without an appropriate lock | 1 |

**Table 3.** Types of concurrency faults.

[from L. Ryzhyk *et al.*, “Dingo: Taming device drivers”, Proc. EuroSys 2009, DOI 10.1145/1519065.1519095]

CC BY ND

# Improving Device Drivers – Engineering

- Model device drivers in object-oriented manner
  - Device drivers generally fit some hierarchy
  - Use object-oriented language; encode common logic into a superclass, instantiated by device-specific subclasses that encode hardware details
    - May be able to encode protocol state machines in the superclass, and leave the details of the hardware access to subclasses (e.g., for Ethernet or USB drivers)
    - May be able to abstract some of the details of the locking, if hardware similar enough
  - Might require multiple inheritance or mixins to encode all the details of the hardware, especially for multi-function devices
- Implementation choices – device driver framework
  - Linux kernel implements this model in C, with much boilerplate
  - MacOS X uses restricted subset of C++ within kernel – simplifies driver development by encoding high-level semantics within framework, leaves only device-specific details to individual drivers

Apple, Inc. “I/O Kit Fundamentals”, 2007
http://developer.apple.com/library/mac/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/IOKitFundamentals.pdf

CC BY ND

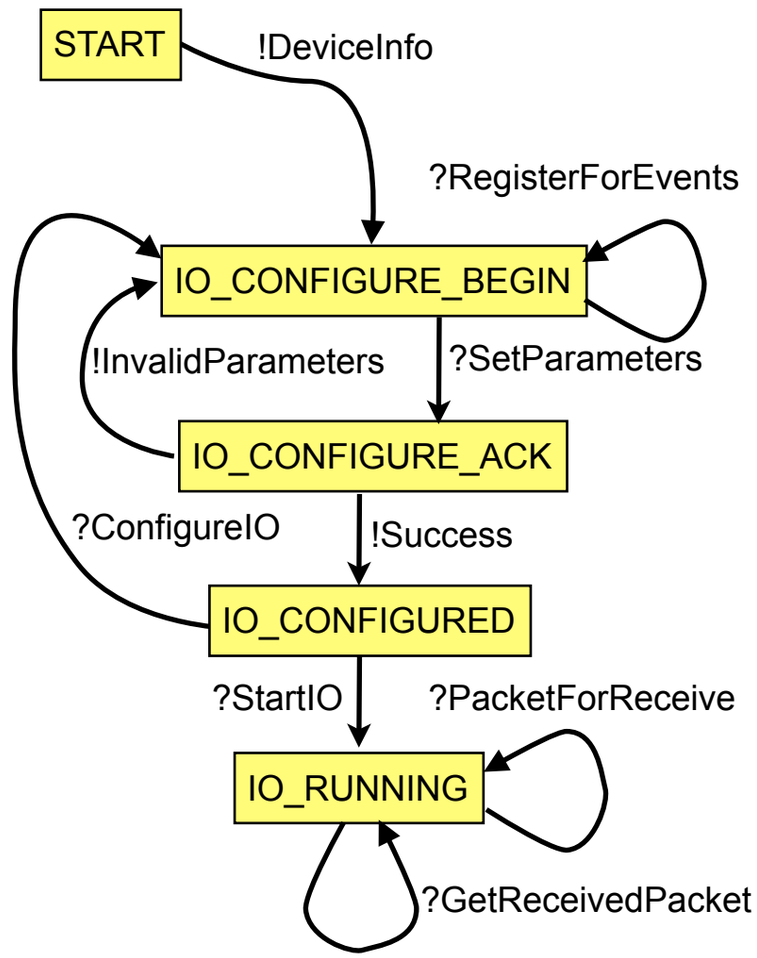
# Improving Device Drivers – State Models

- An ad-hoc device driver model is common
  - Many bugs due to poor specification and documentation of the model
  - Use of object-oriented languages can improve this somewhat, but need careful integration into C-based kernels
- Possible to formalise design as a state machine
  - Make underlying state machine visible in the implementation – MacOS X I/O Kit models incoming events, but not the states, allowable transitions, or generated events
  - Could formally define full state machine in source code, allow automatic verification that driver implements the state machine for its device class, and model checking of the state machine
    - Can be implemented within existing languages, by annotating the code
    - Fits better with more sophisticated, strongly-typed, languages, that can directly model system

BY ND

# Improving Device Drivers – State Models

Example: the Singularity operating system from Microsoft Research




```
contract NicDevice {
  out message DeviceInfo(...);
  in  message RegisterForEvents(NicEvents.Exp:READY
  c);
  in  message SetParameters(...);
  out message InvalidParameters(...);
  out message Success();
  in  message StartIO();
  in  message ConfigureIO();
  in  message PacketForReceive(byte[] in ExHeap p);
  out message BadPacketSize(byte[] in ExHeap p, int
  m);
  in  message GetReceivedPacket();
  out message ReceivedPacket(Packet * in ExHeap p);
  out message NoPacket();

  state START: one {
    DeviceInfo! → IO_CONFIGURE_BEGIN;
  }
  state IO_CONFIGURE_BEGIN: one {
    RegisterForEvents? →
      SetParameters? → IO_CONFIGURE_ACK;
  }
  state IO_CONFIGURE_ACK: one {
    InvalidParameters! → IO_CONFIGURE_BEGIN;
    Success! → IO_CONFIGURED;
  }
  state IO_CONFIGURED: one {
    StartIO? → IO_RUNNING;
    ConfigureIO? → IO_CONFIGURE_BEGIN;
  }
  state IO_RUNNING: one {
    PacketForReceive? → (Success! or BadPacketSize!)
      → IO_RUNNING;
    GetReceivedPacket? → (ReceivedPacket! or
    NoPacket!)
      → IO_RUNNING;
    ...
  }
}
```

**Listing 1. Contract to access a network device driver.**

G. Hunt and J. Larus. “Singularity: Rethinking the software stack”, ACM SIGOPS OS Review, 41(2), April 2007. DOI:10.1145/1243418.1243424

CC BY ND

# Improving Device Drivers – Discussion

- Focus on low-level implementation techniques and high-performance in many device driver models
- Not necessarily appropriate in embedded systems?

- Raising level of abstraction can reduce error-prone boilerplate, allow compiler to diagnose problems

BY ND

# Challenges: Correctness and Security

- Systems may be safety or security critical
- Might need to run for many years, in environments where failures either cause injury or are expensive to fix
  - Medical devices, automotive or flight control, industrial machinery
  - Can you guarantee your system will run for 10 years without crashing?
  - Do you check all the return codes and handle all errors?
  - Fail gracefully?
- Security vulnerabilities in networked systems
  - Privacy and confidentiality – both of data in transit, and against attacks from the network
  - Any networked service is a potential security risk

BY ND

# Evolving Systems Software

- How to address these challenges?
  - Alternative programming models – better languages and tools
  - Alternative kernel designs and system architectures

Colin Perkins | https://csperkins.org/ | Copyright © 2018

# Alternative Programming Models

- Move away from C as an implementation language
  - Lack of type- and memory-safety leads to numerous bugs and security vulnerabilities
  - Limited support for concurrency – race conditions, locking problems – makes it unsuitable for modern machine architectures

- Move towards architectures with a minimal kernel, and strong isolation between other components of the operating system
  - The monolithic part of a kernel is a single failure domain; this needs to be reduced to a minimum → microkernel architecture
  - Easier to debug and manage components when they're isolated from each other, and communicate only through well-defined channels

CC BY ND

# Type- and Memory-Safe Languages

- Type safe language → protects its abstractions
  - Undefined behaviour prohibited by compiler/type system
  - The language specification can require that array bounds are respected, specify the error response to violation, etc.
  - More sophisticated type systems can catch more complex errors – e.g., enforce a socket is connected, check that an input string is correctly escaped to avoid SQL injection, …
- Requires both compile- and run-time checking
  - The *type system* specifies legal properties of the program "for proving the absence of certain program behaviours"
  - Some properties can be statically checked by a compiler: a faulty program will not compile until the bug is fixed
  - Some properties require run-time checks: failure causes a controlled error
  - Doesn't guarantee system works correctly, but ensures it fails in a predictable and consistent way
- Doesn't require byte-code virtual machine; can have efficient implementation

```
-->cat tst.c
#include <stdio.h>

int main()
{
  int x;
  int a;
  int b[13];
  int c;

  a = 1;
  c = 2;

  for (x = 0; x <= 13; x++) {
    b[x] = x;
  }

  printf("a = %d\n", a);
  printf("c = %d\n", c);

  return 0;
}
-->gcc -std=c99 tst.c -o tst
-->./tst
a = 1
c = 13
-->
```

CC BY ND

# Modularity and Microkernels

- Desirable to separate components of a system, so failure of a component doesn't cause failure of the entire system
- Microkernel operating system
  - Strip-down monolithic kernel to essential services; run everything else in user space communicating via message passing API
    - This includes devices drivers, network stack, etc.
    - Kernel just managing process scheduling, isolation, and message passing
  - Widely used in embedded systems, where robustness and flexibility to run devices for unusual hardware are essential features
  - But typically poor performance: frequent context switches expensive, due to need to cross kernel-user space boundary, manage memory protection, etc.

CC BY ND

# Strongly Isolated Systems

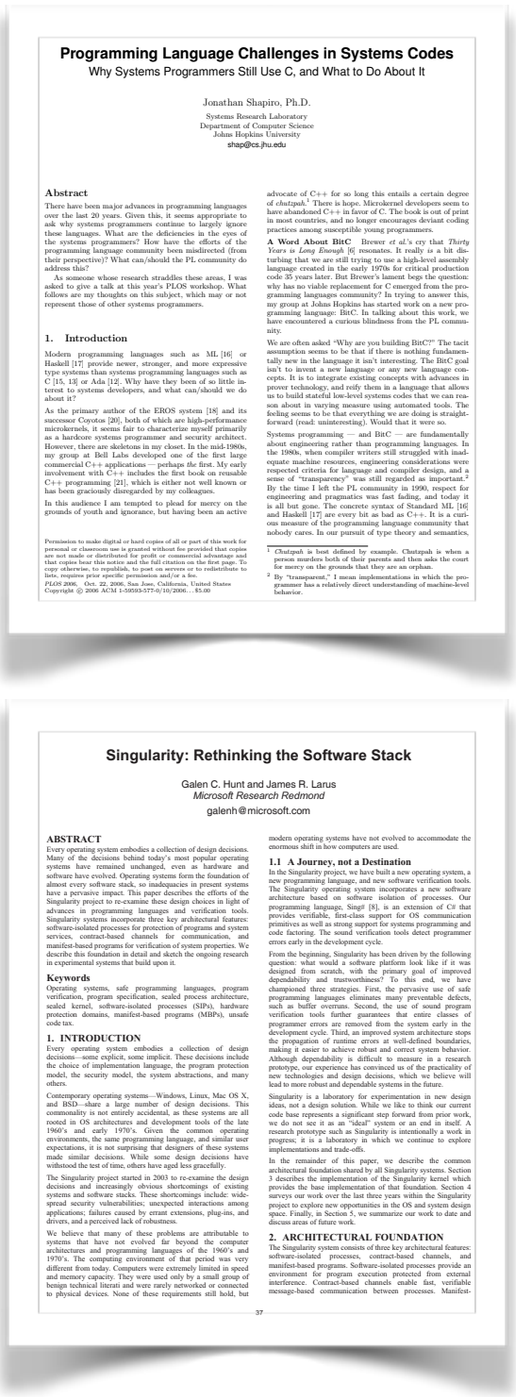
- A possible solution:
  - Microkernel that enforces all code written in a safe language (e.g., by only executing byte code, no native code)
    - This includes device drivers and system services running outside the microkernel
  - Type system prevents malicious code obtaining extra permissions by manipulating memory it doesn't own – done entirely in software; no need to use MMU to enforce process separation
  - A software isolated message passing process architecture – loosely coupled and well suited to multicore hardware
  - Example: the Singularity operating system from Microsoft Research

- Relies on modern programming language features
  - Combination is novel, but individual pieces are well understood

CC BY ND

# Discussion and Further Reading

- Systems software has unique constraints
  - Correctness, robustness, security, performance
  - Low-level programming model was necessary for efficiency – are there alternative models for modern systems?



**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,

[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.

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

**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-



- Further reading:
  - J. Shapiro, "Programming language challenges in systems codes: why systems programmers still use C, and what to do about it", Proc. 3rd workshop on Programming Languages and Operating Systems, San Jose, CA, October 2006. DOI:10.1145/1215995.1216004
  - G. Hunt and J. Larus. "Singularity: Rethinking the software stack", ACM SIGOPS OS Review, 41(2), April 2007. DOI:10.1145/1243418.1243424
  - Read these before the tutorial next week – come prepared to discuss

CC BY ND