

Future Directions

Advanced Systems Programming (H/M)

Lecture 10

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow | 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.

Systems Programming

- Course has discussed advanced topics in systems programming:
  - What is systems programming?
  - Type systems
  - Type-based modelling and design
  - Resource ownership and memory management
  - Garbage collection
  - Concurrency
  - Coroutines and asynchronous programming
  - Security considerations

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Key Lessons

- Systems programs need low-level control of data layout
  - Device drivers
  - Network protocols
  - File I/O
- This control does not need to come at the expense of safety – modern languages can provide control
  - Memory safety with control over data layout
  - Avoidance of use-after-free, data races, iterator invalidation
- Safe concurrency is important, but all approaches have trade-offs
- Type systems can help model the problem domain
  - Model the assumptions – correctness and security

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Future Directions

BY ND

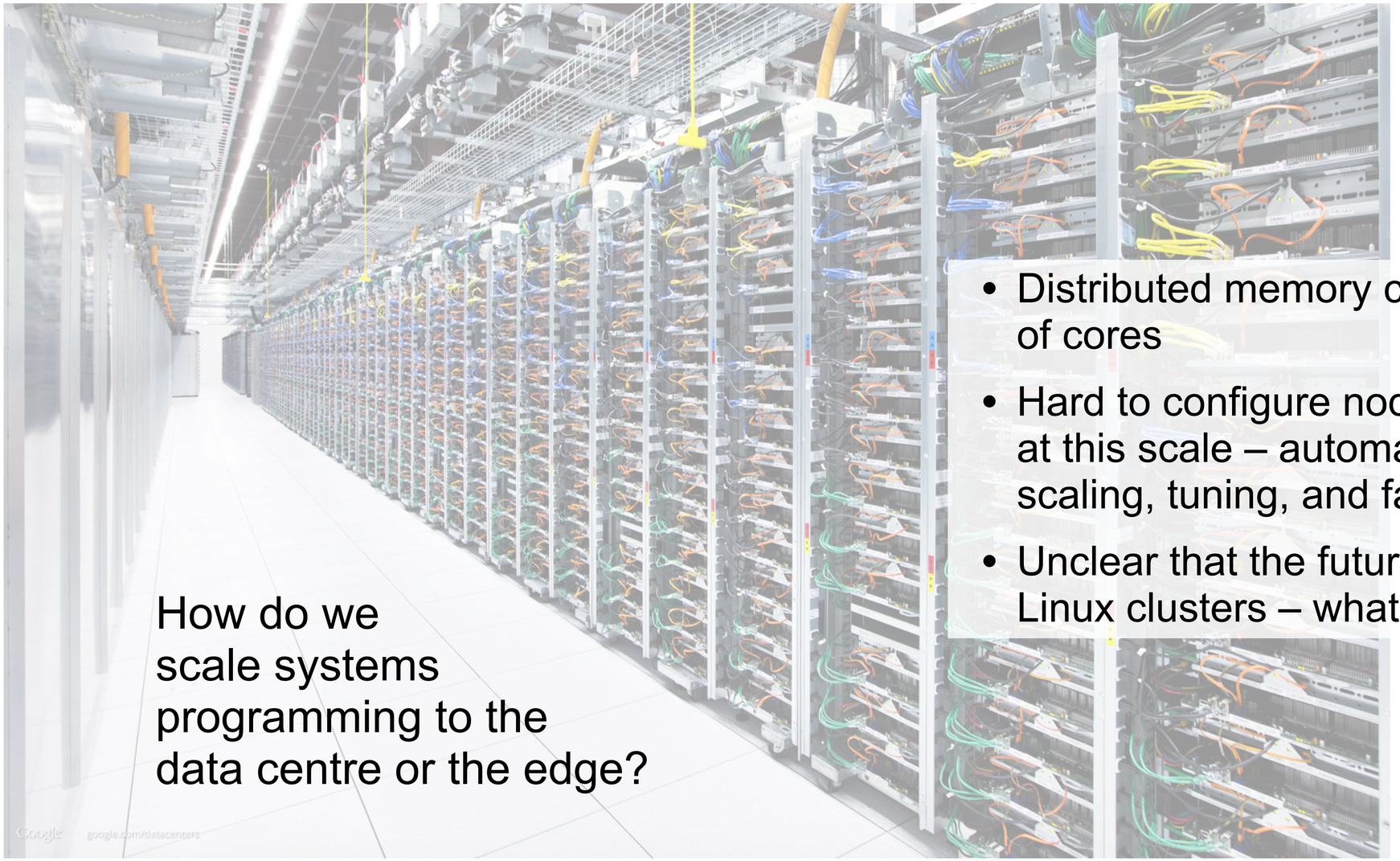
Concurrency

- Still no good solutions for concurrent programming
  - Transactional memory doesn't sit well with impure imperative languages
  - Message passing has issues with back pressure, race conditions, deadlocks, large-scale orchestration
  - Asynchronous code introduces subtle blocking and starvation bugs
  - Unclear what is the right solution

- None of these approaches scale to GPU programming → implicit parallelism? data parallel array types?

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Data Centre Scale Computing



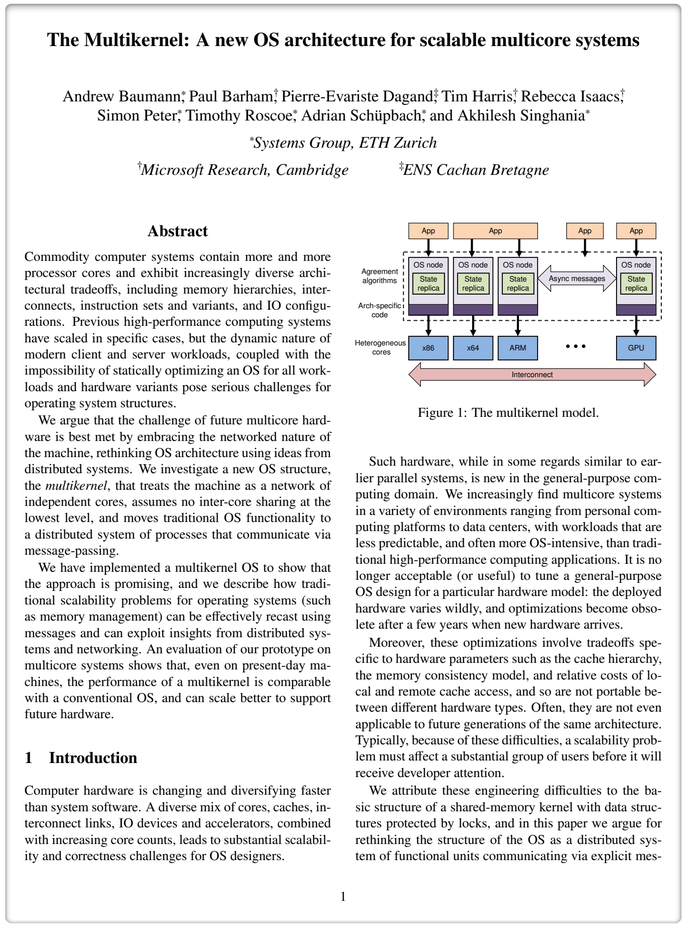
How do we scale systems programming to the data centre or the edge?

- Distributed memory computation on millions of cores
- Hard to configure nodes and communications at this scale – automatic configuration, scaling, tuning, and fault tolerance essential
- Unclear that the future should be large-scale Linux clusters – what comes next?

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Distributed Operating Systems

- Barrelfish – a message passing kernel for multicore systems
  - Each core runs a separate operating systems instance
  - **All** communication between cores is via message passing; no “main” CPU to act as central orchestrator
- A research prototype to test an idea, not a product
  - Where is the boundary for a Barrelfish-like system?
  - Distinction between a distributed multi-kernel and a distributed system of networked computers? Should there be such a distinction?
  - How does it relate to concurrent programming, data centres, distributed edge computing, etc?



The Multikernel: A new OS architecture for scalable multicore systems

Andrew Baumann*, Paul Barham†, Pierre-Evariste Dagand‡, Tim Harris†, Rebecca Isaacs†, Simon Peter*, Timothy Roscoe*, Adrian Schüpbach*, and Akhilesh Singhania*

*Systems Group, ETH Zurich

†Microsoft Research, Cambridge ‡ENS Cachan Bretagne

Abstract

Commodity computer systems contain more and more processor cores and exhibit increasingly diverse architectural tradeoffs, including memory hierarchies, interconnects, instruction sets and variants, and IO configurations. Previous high-performance computing systems have scaled in specific cases, but the dynamic nature of modern client and server workloads, coupled with the impossibility of statically optimizing an OS for all workloads and hardware variants pose serious challenges for operating system structures.

We argue that the challenge of future multicore hardware is best met by embracing the networked nature of the machine, rethinking OS architecture using ideas from distributed systems. We investigate a new OS structure, the *multikernel*, that treats the machine as a network of independent cores, assumes no inter-core sharing at the lowest level, and moves traditional OS functionality to a distributed system of processes that communicate via message-passing.

We have implemented a multikernel OS to show that the approach is promising, and we describe how traditional scalability problems for operating systems (such as memory management) can be effectively recast using messages and can exploit insights from distributed systems and networking. An evaluation of our prototype on multicore systems shows that, even on present-day machines, the performance of a multikernel is comparable with a conventional OS, and can scale better to support future hardware.

Figure 1: The multikernel model.

1 Introduction

Computer hardware is changing and diversifying faster than system software. A diverse mix of cores, caches, interconnect links, IO devices and accelerators, combined with increasing core counts, leads to substantial scalability and correctness challenges for OS designers.

Such hardware, while in some regards similar to earlier parallel systems, is new in the general-purpose computing domain. We increasingly find multicore systems in a variety of environments ranging from personal computing platforms to data centers, with workloads that are less predictable, and often more OS-intensive, than traditional high-performance computing applications. It is no longer acceptable (or useful) to tune a general-purpose OS design for a particular hardware model: the deployed hardware varies wildly, and optimizations become obsolete after a few years when new hardware arrives.

Moreover, these optimizations involve tradeoffs specific to hardware parameters such as the cache hierarchy, the memory consistency model, and relative costs of local and remote cache access, and so are not portable between different hardware types. Often, they are not even applicable to future generations of the same architecture. Typically, because of these difficulties, a scalability problem must affect a substantial group of users before it will receive developer attention.

We attribute these engineering difficulties to the basic structure of a shared-memory kernel with data structures protected by locks, and in this paper we argue for rethinking the structure of the OS as a distributed system of functional units communicating via explicit mes-

1

A. Baumann *et al.*, “The Multikernel: A new OS architecture for scalable multicore systems”, ACM Symposium on Operating Systems Principles, 2009. DOI:10.1145/1629575.1629579

CC BY ND

Future Systems Programming Languages

- Increasingly seeing research ideas incorporated into mainstream programming languages
  - Rust, Swift – abstraction, resource management, strong and expressive type systems, low-level control
  - Erlang, Go – concurrency, fault tolerance, communication









- Choose the language based on problem domain, available tooling, and expertise
  - Good solutions for programming single systems
  - We still have a lot to learn about scaling to large clusters

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Scaling Systems Programming





- Unix and C made sense when programming simple single-core systems
  - And form the basis of most existing systems – **but what comes next?**
- Modern massively concurrent and distributed applications need new approaches:
  - Low-level performance still crucial, but also need type systems and abstractions to hide the complexity – *no-one* can hold all the details in their head
  - Tooling is essential – to check invariants, document assumptions, ensure consistency
  - Deployment, configuration, and management must become increasingly autonomic – DevOps, infrastructure as code, self-managing

CC BY ND

Wrap-Up

BY ND

Assessment

- Marks for assessed coursework will be available shortly

- Final exam, worth 80% of the marks for the course, will be held in April/May
  - Sample exam paper, with worked answers, is available
  - Material from the lectures, labs, 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

- No specific revision lecture – use the Teams chat, or email, for revision questions

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

The End



;login: logout

The Night Watch

JAMES MICKENS

James Mickens is a researcher in the Distributed Systems group at Microsoft's Redmond lab. His current research focuses on web applications, with an emphasis on the design of JavaScript frameworks that allow developers to diagnose and fix bugs in widely deployed web applications. James also works on fast, scalable storage systems for datacenters. James received his PhD in computer science from the University of Michigan, and a bachelor's degree in computer science from Georgia Tech. mickens@microsoft.com

As a highly trained academic researcher, I spend a lot of time trying to advance the frontiers of human knowledge. However, as someone who was born in the South, I secretly believe that true progress is a fantasy, and that I need to prepare for the end times, and for the chickens coming home to roost, and fast zombies, and slow zombies, and the polite zombies who say "sir" and "ma'am" but then try to eat your brain to acquire your skills. When the revolution comes, I need to be prepared; thus, in the quiet moments, when I'm not producing incredible scientific breakthroughs, I think about what I'll do when the weather forecast inevitably becomes RIVERS OF BLOOD ALL DAY EVERY DAY. The main thing that I ponder is who will be in my gang, because the likelihood of post-apocalyptic survival is directly related to the size and quality of your rag-tag group of associates. There are some obvious people who I'll need to recruit: a locksmith (to open doors); a demolitions expert (for when the locksmith has run out of ideas); and a person who can procure, train, and then throw snakes at my enemies (because, in a world without hope, snake throwing is a reasonable way to resolve disputes). All of these people will play a role in my ultimate success as a dystopian warlord philosopher. However, the most important person in my gang will be a systems programmer. A person who can debug a device driver or a distributed system is a person who can be trusted in a Hobbesian nightmare of breathtaking scope; a systems programmer has seen the terrors of the world and understood the intrinsic horror of existence. The systems programmer has written drivers for buggy devices whose firmware was implemented by a drunken child or a sober goldfish. The systems programmer has traced a network problem across eight machines, three time zones, and a brief diversion into Amish country, where the problem was transmitted in the front left hoof of a mule named Deliverance. The systems programmer has read the kernel source, to better understand the deep ways of the universe, and the systems programmer has seen the comment in the scheduler that says "DOES THIS WORK LOL," and the systems programmer has wept instead of LOLed, and the systems programmer has submitted a kernel patch to restore balance to The Force and fix the priority inversion that was causing MySQL to hang. A systems programmer will know what to do when society breaks down, because the systems programmer already lives in a world without law.

J. Mickens. The night watch. ;login: logout, pages 5–8, November 2013. https://www.usenix.org/system/files/1311_05-08_mickens.pdf

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow