University of Glasgow | School of Computing Science

Wrap-up and Review

Advanced Operating Systems (M)
Lecture 20

Lecture Outline

- Review of material
- Conclusions and Future Directions
- Examination

Review of Material

- Unix/Linux and Windows are the outcome of a long strand of operating systems development
  - The C programming language
  - Monolithic kernels
    - Unix – unbroken line of evolution since the early 1970s
    - Linux – reimplementation of Unix ideas, for the 1990s
    - Windows – builds on Digital Equipment Corporation VAX/VMS dating from 1975
- Operating systems and programming language research have evolved since the 1970s – how might this affect future operating systems?

Real-time Operating Systems

- Introduction to real-time systems
- Real-time scheduling
  - Clock driven scheduling
  - Priority driven scheduling:
    - Periodic, aperiodic and sporadic tasks
    - Rate and deadline monotonic scheduling, earliest deadline first, least slack time
  - Proofs of correctness
    - Maximum utilisation tests, time demand analysis
- Resource access control
  - Priority inheritance protocol; priority ceiling protocol; impact of scheduling
- Implementation techniques
  - Real-time APIs and code; implementing real-time schedulers

Systems Programming

- Programming real-time and embedded systems
  - Interacting with hardware
  - Interrupt and timer latency
  - Memory issues
  - Power, size and performance constraints
- System longevity
- Development and debugging

- Traditional approaches; possible future alternatives
  - Moving beyond C for the embedded world

Dependable Device Drivers

- Sources of bugs in device drivers
- Engineering approaches to improving device driver reliability
  - Use of object-oriented code and languages for device drivers
  - MacOS X I/O Kit as a example
- Future directions: explicit identification of driver state machines
  - Formal verification driver code
  - Integration with model checking
  - Dingo and Singularity as examples

Dependable Kernels

- Evolution of the operating system kernel
  - Microkernels
  - Use of managed code for systems programming – how much of the kernel can be written in a high-level type-safe language?
  - Pervasive concurrency
  - Examples: Singularity and BarrelFish

Garbage Collection

- Memory management models
  - Garbage collection – advantages and disadvantages
  - Other approaches – e.g., RAII
- Role of garbage collection in future kernels
- Garbage collection algorithms and their properties
  - Mark-sweep, mark-compact, copying collectors, generational collectors, incremental collectors and tricolour marking, Cheney algorithm
- Real-time garbage collection
  - Specially-tuned incremental collector, treated as a periodic tasks
  - Places limits on the amount of garbage that can be created

Concurrency

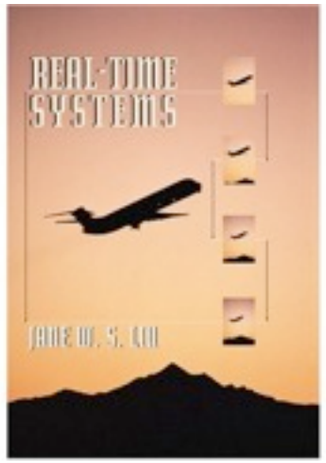
- Pervasive concurrency, and its implications for next generation operating systems
- Software Transactional Memory
  - Transactional processing as the fundamental concurrency primitive
  - Relation to purely functional languages
  - Implementation in Haskell
- Actors and message passing
  - Exchange of immutable messages between concurrent processes as the fundamental concurrency primitive
  - Implementation in Erlang and Singularity
  - Robustness – “let it crash”

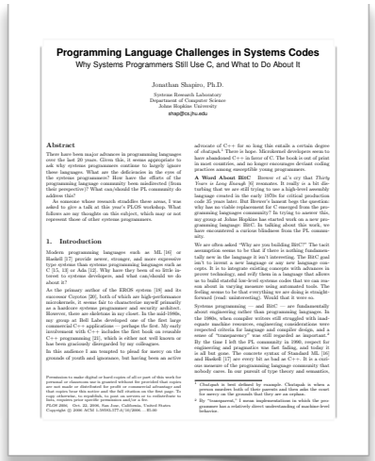
Discussion

- Wide spectrum of research ideas and concepts
- Which are seeing widespread use?
  - Functional languages and message passing concurrency
  - Garbage collection – potential for integration with kernels
  - Increased use of static code analysis tools, to debug the limitations of C
- Opportunities for dependable kernels
  - New implementation frameworks and safe programming languages
  - Approaches similar to Singularity have large potential

Discussion

What are the key ideas that emerge from the papers and discussion?




Programming Language Challenges in Systems Codes
Why Systems Programmers Still Use C, and What to Do About It

Dingo: Taming Device Drivers

A Real-time Garbage Collector with Low Overhead and Consistent Utilization

Hera-JVM: A Runtime System for Heterogeneous Multi-Core Architectures


Thirty Years is Long Enough: Getting Beyond C

Sealing OS Processes to Improve Dependability and Safety

The Multikernel: A new OS architecture for scalable multicore systems

Composable Memory Transactions


Singularity: Rethinking the Software Stack

Uniprocessor Garbage Collection Techniques

Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses


contributed articles

Erlang

Examination

- Weighting: 80%
- Duration 2 hours
  - Sample paper will be made available on Moodle
  - Old RTES(M) exams provide good sample questions for the real-time scheduling material
- Material covered in the lectures, tutorials, and 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

End