Evolution of Systems Programming

Advanced Operating Systems (M)
Lecture 11
Real-time and Embedded Programming

- Real time and embedded systems differ from conventional desktop applications
  - Must respect timing constraints – scheduling theory in prior lectures
  - Must interact with hardware and the environment
  - Often very sensitive to correctness and robust operation
  - Often very sensitive to cost, weight, or power consumption

- Implications to consider:
  - Proofs of correctness, scheduling tests, etc.
  - Limited resources: low level programming environments; high awareness of systems issues; interaction with hardware
  - Challenges imposed on operating system and programming environment by resource constraints and programming model
Yes, but...

• Continued advances in hardware, supporting both traditional embedded systems and new ubiquitous computing platforms
  • Moore’s “law” shows no sign of abating

• Where are corresponding advances in software?
  • Desirable to raise abstraction level: ease program development and increase productivity, employ modern software engineering techniques and high(er) level languages
  • Simplify proofs of correctness
Evolution of Systems Programming

• Use increased system performance to provide:
  • Language and runtime support for low-level programming: interrupt handling; device access; etc.
  • Language and runtime support for automatic memory management, including real-time garbage collection
  • Language and runtime support for real-time systems: periodic threads; timed statements/timing annotations
  • Language and runtime support for concurrency: type systems to ensure correctness; message passing; transactional memory

• Emphasis on real-time, embedded, and ubiquitous systems
  • iOS and Android begin to show the possibilities – but, what next?
Low-level Programming: Device Access

• Various approaches to low-level hardware access
  • C-style: simple and expressive, non-portable
  • Ada: verbose, precise specification, portable

• Can language and runtime support help?
  • Well-defined integral types and easy support for bit manipulation desirable
  • Clear that object-oriented ideas useful for device driver families:
    • MacOS X I/O Kit – object oriented device drivers using a subset of C++ (without exceptions, multiple inheritance, templates, RTTI)
    • Linux uses object-based approach for many drivers, implemented in C: higher performance, but MacOS X drivers easier to write
    • Simple object-oriented extensions to C to define sub-class relationships that can abstract out common behaviour would provide great benefit

• Better ways to represent state machines, timeouts, and interrupt handlers at language level likely beneficial → concurrency and real-time support
Low-level Programming: Interrupt Handling

- Interrupt handling system dependent
  - Few systems support linking user code into interrupt handlers
  - Ada real-time systems annex a notable exception:

```
package Ada.Interrupts is
  type Interrupt_Id is ...;
  type Parameterless_Handler is access protected procedure;

  function Is_Reserved(Interrupt:Interrupt_Id) return Boolean;
  function Is_Attached(Interrupt:Interrupt_Id) return Boolean;
  function Current_Handler(Interrupt:Interrupt_Id) return Parameterless_Handler;
  procedure Attach_Handler(Handler:Parameterless_Handler, Interrupt:Interrupt_Id);
  procedure Detach_Handler(Interrupt:Interrupt_Id);
  ...
end Ada.Interrupts;
```

- Could provide similar standard facilities in other languages
  - Must vector through hardware abstraction layer and kernel, but relatively straightforward to implement as a standard library for adding interrupt handlers to a microkernel OS
  - Could eliminate platform-specific hooks, allow portable code
  - More interesting: interaction with message-passing concurrency mechanisms
Automatic Memory Management

• Real-time systems community has a strong distrust of automatic memory management
  • E.g., the real-time extensions to Java augmented the memory model with non-garbage collected regions and manual memory management
  • But, memory management problems abound
    • Memory leaks and unpredictable memory allocation performance (calls to `malloc()` can vary in execution time by several orders of magnitude)
    • Memory corruption and buffer overflows

• Can automatic memory management be provided that satisfies the real-time systems community?
  • Predictable, low-overhead, real-time garbage collection
  • Languages with type systems that can control resource management or enforce access controls without hardware memory protection
    • The RAII idiom in C++ or `using` in C# give hints in this direction
Garbage Collection

• Traditional algorithms not suitable

  • Triggered at unpredictable times; unpredictable collection delays as data is moved to avoid heap fragmentation

• Active research into real-time garbage collection

  • Two basic approaches:
    • Work based: every request to allocate an object or assign an object reference does some garbage collection; amortise collection cost with allocation cost
    • Time based: schedule an incremental collector as a periodic task

  • Obtain timing guarantees only by limiting amount of garbage that can be collected in a given interval

    • Implication: user must indicate maximum memory consumption and allocation rate, to determine cost of the garbage collector
    • Workable solutions exist for many periodic real-time applications; same issue as certain scheduling algorithms placing constraints on application design

Memory Protection

• Traditional memory protection is unpredictable
  • Slows context switch and system call times due to managing page tables
  • Requires illegal access traps and error handlers: difficult to implement

• Can guarantee safety without hardware protection
  • Strongly typed language, checked array bounds, no pointer arithmetic: looks more like Java than C
    • Difficulty is in efficient representation of data, and handling aliasing of memory regions
    • Examples: BitC, Cyclone
  • Much verification done at compile time; reduces run-time unpredictability
  • Example: Singularity operating system from Microsoft Research
    • Mostly written in extended C#, small microkernel in C++; language ensures that inter-process communication is done via strongly-typed message passing; no hardware memory protection
    • http://research.microsoft.com/os/singularity/
Timing and Real-time Systems

• How to ensure predictable timing?
  • Theory of real-time scheduling well-developed, provided requirements are clearly specified
  • Introduce abstractions for periodic threads into the language and runtime support system
    • E.g., the real-time extensions to Java add RealtimeThread
  
• Add timing annotations, let compiler/runtime validate scheduling proof?
  • Compiler much better at counting cycles than a human on modern processor architectures
  • Likely feasible to estimate worst-case execution time for many embedded codes, which can be compared with task timing annotations
  • Computationally infeasible in the general case (due to loops, etc.) but most real time systems are more constrained: otherwise how can they be manually proved to meet timing bounds?
  • Helps debugging if not proving correctness

```java
class RealtimeThread extends java.lang.Thread {
    // ...additional constructors to specify
    // SchedulingParameters
    ...
    // ...adds additional methods:
    public void setScheduler(Scheduler s);
    public void schedulePeriodic();
    public boolean waitForNextPeriod();
    ...
}
```

Timing Annotations

• Is adding such timing annotations feasible?
  • Properties of periodic tasks straightforward, if expressed in language
  • Aperiodic/sporadic tasks harder, but often meaningful statistics
  • But what about low-level behaviour?
    • Annotate that an expression should take no more than $x$ milliseconds; check generated code
    • Operating system calls and library functions will need to be annotated
  • What are hidden timing behaviours of system?
    • Scheduler and system call overhead
    • `malloc() / free()`, garbage collection
    • Cache, memory hierarchy, memory protection
    • Speculative execution, pipelining, super-scalar and out-of-order execution

• Programmers cannot count cycles; yet many still program as if it were possible – need compiler help
Concurrency increasingly important

- Multicore systems now ubiquitous
- Asynchronous interactions between software and hardware devices

Threads and synchronisation primitives problematic

- Low level; easy to make mistakes; hard to reason about correctness

Are there alternatives that avoid these issues?

- Implicit concurrency; execution models which hide complexity
- Functional and/or message passing algorithms
  - Example: Ericsson AXD301 160 Gbps ATM switch had 99.9999999% uptime and was (mostly) written in the Erlang functional programming language
  - Transactional memory coupled with functional languages (e.g., Haskell) for automatic rollback and retry of transactions


→ Lectures 16-19
Reliability Through Clarity

• State and requirements hidden in existing code
  • Need to infer high-level goals from low-level implementation
  • Yet Moore’s law continues: performance increasing for fixed price point, power consumption

• Better languages and runtime support will allow programmers to express high-level goals, system to check implementation meets them
  • Requires paradigm shift away from current implementation strategies
Further Reading

