Real-time & Embedded Systems Programming

Advanced Operating Systems
Lecture 7
Lecture Outline

• Ensuring predictable timing
• Embedded systems
  • Constraints
  • Interacting with hardware
  • Device drivers
• Correctness and system longevity
• Low-level programming environments
  • Current and alternative approaches
Ensuring Predictable Timing

- Scheduling theory gives proof of correctness – if timing of system well understood
- 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
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
Interacting with Hardware

- Devices represented by bit fields at known address
  - Bit-level manipulation needed to access fields in control register
  - Code needs memory address and size of control register, layout, endianness, and meaning of bit fields within the 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 compiler/operating system defines these for its environment; the <stdint.h> and <limits.h> headers provide definitions to help with portability, but with weak type checking

- Device drivers written in C a frequent source of bugs

- Other languages (e.g., Ada) provide strict definitions and allow for stronger type checking

```c
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
Sources of Bugs in Device Drivers (1)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Total faults</th>
<th>Device prot. violations</th>
<th>S/W protocol violations</th>
<th>Concurrency faults</th>
<th>Generic faults</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>USB drivers</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rtl8150</td>
<td>rtl8150 USB-to-Ethernet adapter</td>
<td>16</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>catc</td>
<td>e1210a USB-to-Ethernet adapter</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>kaweth</td>
<td>K5kusb101 USB-to-Ethernet adapter</td>
<td>15</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>usb net</td>
<td>generic USB network driver</td>
<td>45</td>
<td>16</td>
<td>9</td>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>usb hub</td>
<td>USB hub</td>
<td>67</td>
<td>27</td>
<td>16</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>usb serial</td>
<td>USB-to-serial converter</td>
<td>50</td>
<td>2</td>
<td>17</td>
<td>13</td>
<td>18</td>
</tr>
<tr>
<td>usb storage</td>
<td>USB Mass Storage devices</td>
<td>23</td>
<td>7</td>
<td>5</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td><strong>IEEE 1394 drivers</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eth1394</td>
<td>generic ieee1394 Ethernet driver</td>
<td>22</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>sbp2</td>
<td>sbp-2 transport protocol</td>
<td>46</td>
<td>18</td>
<td>10</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td><strong>PCI drivers</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mthca</td>
<td>InfiniHost InfiniBand adapter</td>
<td>123</td>
<td>52</td>
<td>22</td>
<td>11</td>
<td>38</td>
</tr>
<tr>
<td>bnx2</td>
<td>bnx2 network driver</td>
<td>51</td>
<td>35</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>i810 fb</td>
<td>i810 frame buffer device</td>
<td>16</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>cmipci</td>
<td>cmi8338 soundcard</td>
<td>22</td>
<td>17</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>498</td>
<td>189 (38%)</td>
<td>101 (20%)</td>
<td>93 (19%)</td>
<td>115 (23%)</td>
<td></td>
</tr>
</tbody>
</table>

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?


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.
Sources of Bugs in Device Drivers (2)

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

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

<table>
<thead>
<tr>
<th>Type of faults</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordering violations</td>
<td></td>
</tr>
<tr>
<td>Driver configuration protocol violation</td>
<td>16</td>
</tr>
<tr>
<td>Data protocol violation</td>
<td>9</td>
</tr>
<tr>
<td>Resource ownership protocol violation</td>
<td>8</td>
</tr>
<tr>
<td>Power management protocol violation</td>
<td>8</td>
</tr>
<tr>
<td>Hot unplug protocol violation</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Format violations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect use of OS data structures</td>
<td>29</td>
</tr>
<tr>
<td>Passing an incorrect argument to an OS service</td>
<td>19</td>
</tr>
<tr>
<td>Returning invalid error code</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 2. Types of software protocol violations.

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

Table 3. Types of concurrency faults.

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 the hardware is 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

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
Improving Device Drivers – State Models

Example: the Singularity operating system from Microsoft Research

```
contract NicDevice {
    out message DeviceInfo(...);
    in message RegisterForEvents(NicEvents.Exp:READY);
    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.
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
Correctness and System Longevity

• Systems may be safety critical or difficult to update
  • Medical devices, automotive or flight control, industrial machinery
  • DVD player, washing machine, microwave oven, car engine controller

• Might need to run for many years, in environments where failures either cause injury or are expensive to fix
  • 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?
Low-level Programming Environment

• Embedded systems often 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
Development and Debugging

• Systems may be too limited to run compiler
  • Develop using a cross compiler running on a PC, download code using a serial line, or by burning a flash ROM and installing

• May have limited debugging facilities:
  • Serial line connection to host PC
  • LEDs on the development board
  • Logic analyser or other hardware test equipment
  • Formal proofs of correctness become more attractive when real system so difficult to analyse…
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
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 on web forms...

- **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**
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.
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
Discussion

• Real-time and embedded systems have unique constraints; strong correctness concerns

• Low-level programming model was necessary for efficiency – alternatives for modern systems?

• 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
  • Both papers will be discussed in tutorial 3