Device Drivers

COMP9242
2009/S2 Week 9
Some statistics

- 70% of OS code is in device drivers
  - 3,448,000 out of 4,997,000 loc in Linux 2.6.27

- A typical Linux laptop runs ~240,000 lines of kernel code, including ~72,000 loc in 36 different device drivers

- Drivers contain 3—7 times more bugs per loc than the rest of the kernel

- 70% of OS failures are caused by driver bugs
Lecture outline

- Part 1: Introduction to device drivers
- Part 2: Overview of research on device driver reliability
- Part 3: Device drivers research at ERTOS
Part 1: Introduction to device drivers
Functions of a driver

• Encapsulation
  - Hides low-level device protocol details from the client

• Unification
  - Makes similar devices look the same

• Protection (in cooperation with the OS)
  - Only authorised applications can use the device

• Multiplexing (in cooperation with the OS)
  - Multiple applications can use the device concurrently
OS archeology

The first (?) device drivers: I/O libraries for the IBM 709 batch processing system [1958]
OS archeology

The first (?) device drivers: I/O libraries for the IBM 709 batch processing system [1958]

Protection: prevent a user program from corrupting data belonging to the supervisor or to other programs
IBM 7090 [1959] introduced I/O channels, which allowed I/O and computation to overlap
IBM 7090 [1959] introduced I/O channels, which allowed I/O and computation to overlap.

“the complex routines were required to allow even the simplest user program to take full advantage of the hardware, but writing them was beyond the capability of the majority of programmers.”

IBM 7094 [1962] supported a wide range of peripherals: tapes, disks, teletypes, flexowriters, etc.
IBM 7094 [1962] supported a wide range of peripherals: tapes, disks, teletypes, flexowriters, etc.
OS archeology

GE-635 [1963] introduced the master CPU mode. Only the hypervisor running in the master mode could execute I/O instructions
I/O devices in a typical desktop system
PCI bus overview

- **PCI bus**
  - Conventional PCI
    - Developed and standardised in early 90's
    - 32 or 64 bit shared parallel bus
    - Up to 66MHz (533MB/s)
  - PCI-X
    - Up to 133MHz (1066MB/s)
  - PCI Express
    - Consists of serial p2p links
    - Software-compatible with conventional PCI
    - Up to 16GB/s per device
PCI bus overview: memory space

Physical address space (FSB)

PCI controller

RAM

PCI memory space

Dev1  Dev2  Dev3

Dev1  Dev2  Dev3
PCI bus overview: DMA

CPU -> PCI controller

PCI controller -> Physical address space (FSB)

PCI controller -> RAM

PCI controller -> PCI memory space

Dev1 -> PCI controller
Dev2 -> PCI controller
Dev3 -> PCI controller
PCI bus overview: config and I/O spaces

- PCI configuration space
  - Used for device enumeration and configuration
  - Contains standardised device descriptors

- I/O space
  - obsolete
USB bus overview

- USB bus
  - Host-centric
  - Distributed-system-style architecture
  - Hot plug
  - Power management
    - Bus-powered and self-powered devices
- USB 1.x
  - Up to 12Mb/s
- USB 2.0
  - Up to 480Mb/s
- USB 3.0
  - Up to 4.8Gb/s
USB bus overview

- Root hub
  - Device 3
  - Device 4
    - Device 2
    - Device 1
      - Hub
        - DMA
          - USB bus controller
            - DMA
              - Transfer descriptors
                - Completions
I/O devices in a typical desktop system

- CPU
- Memory controller
- Interrupt controller
- Front-side bus
- PCI bus controller (ICH7)
- Ethernet controller (BCM4401)
- Graphics controller (GMA950)
- USB bus
- USB controller (EHCI)
- SATA bus controller (AHCI)
- USB Ethernet controller (AX88772)
- SATA bus
- SATA disk
Driver stacking

A diagram showing the architecture of a computer system, including the CPU, Memory controller, PCI bus controller (ICH7), USB controller (EHCI), USB Ethernet controller (AX88772), and Front-side bus connections.
Driver stacking

TCP/IP stack

AX88772 Ethernet driver

hard_start_xmit(pkt)
Driver stacking

CPU

Memory controller

PCI bus controller (ICH7)

USB bus

USB controller (EHCI)

USB Ethernet controller (AX88772)

USB EHCI controller driver

AX88772 Ethernet driver

USB submit urb(urb)

TCP/IP stack

hard_start_xmit(pkt)
Driver stacking

CPU

Memory controller

PCI bus controller (ICH7)

USB bus

USB Ethernet controller (AX88772)

Front-side bus

USB controller (EHCI)

USB EHCI controller driver

TCP/IP stack

hard_start_xmit(pkt)

AX88772 Ethernet driver

usb_submit_urb(urb)

PCI bus driver

mem loads/stores
Driver stacking

- Memory controller
- USB controller (EHCI)
- USB Ethernet controller (AX88772)
- PCI bus controller (ICH7)
- AX88772 Ethernet driver
- USB EHCI controller driver
- PCI bus driver
- PCI framework
- USB framework
- TCP/IP stack

Front-side bus
Driver framework design patterns

The driver pattern

- Client
- Driver
- Device
  - Device-class interface
  - Bus client interface
    - Bus framework
      - Device protocol
        - Device

The bus pattern

- Client
- Client driver
- Bus framework
  - Client driver
    - Client driver
      - Bus client interface
        - Bus interface
          - Bus driver
Questions?
Part 2: Overview of research on device driver reliability
Some statistics

- 70% of OS code is in device drivers
  - 3,448,000 out of 4,997,000 loc in Linux 2.6.27

- A typical Linux laptop runs ~240,000 lines of kernel code, including ~72,000 loc in 36 different device drivers

- Drivers contain 3—7 times more bugs per loc than the rest of the kernel

- 70% of OS failures are caused by driver bugs
Understanding driver bugs

- Driver failures
Understanding driver bugs

• Driver failures
  - Memory access violations
  - OS protocol violations
    • Ordering violations
    • Data format violations
    • Excessive use of resources
    • Temporal failure
  - Device protocol violations
    • Incorrect use of the device state machine
    • Runaway DMA
User-level device drivers

• User-level drivers
  - Each driver is encapsulated inside a separated hardware protection domain
  - Communication between the driver and its client is based on IPC
  - Device memory is mapped into the virtual address space of the driver
  - Interrupts are delivered to the driver via IPC's
User-level drivers in μ-kernel OSs

- User land
  - Mem-mapped I/O
  - Kernel

- Driver
  - TCP/IP
  - Application

- IPC
User-level drivers in μ-kernel OSs

Diagram showing the interactions between User land, Kernel, Driver, TCP/IP, and Application with IPC and IRQ.
User-level drivers in µ-kernel OSs

- Net filter
- Driver
- Application
  - TCP/IP

User land

Kernel

IPC
User-level drivers in µ-kernel OSa
Driver performance characteristics
Driver performance characteristics

- I/O throughput
  - Can the driver saturate the device?
- I/O latency
  - How does the driver affect the latency of a single I/O request?
- CPU utilisation
  - How much CPU overhead does the driver introduce?
Early implementations

• Michigan Terminal System [1970's]
  - OS for IBM System/360
  - Apparently, the first to support user-level drivers

• Mach [1985-1994]
  - Distributed multi-personality µ-kernel-based multi-server OS
  - High IPC overhead
  - Eventually, moved drivers back into the kernel

• L3 [1987-1993]
  - Persistent µ-kernel-based OS
  - High IPC overhead
  - Improved IPC design: 20-fold performance improvement
  - No data on driver performance available
More recent implementations

- **Sawmill [~2000]**
  - Multiserver OS based on automatic refactoring of the Linux kernel
  - Hampered by software engineering problems
  - No data on driver performance available

- **DROPS [1998]**
  - L4 Fiasco-based real-time OS
  - ~100% CPU overhead due to user-level drivers

- **Fluke [1996]**
  - ~100% CPU overhead

- **Mungi [1993—2006]**
  - Single-address-space distributed L4-based OS
  - Low-overhead user-level I/O demonstrated for a disk driver
Currently active systems

• Research
  - seL4
  - MINIX3
  - Nexus

• Commercial
  - OKL4
  - QNX
  - GreenHills INTEGRITY
Improving the performance of ULD
Ways to improve user-level driver performance
- Shared-memory communication
- Request queueing
- Interrupt coalescing
Implementing efficient shared-memory communication

- Issues:
  - Resource accounting
  - Safety
  - Asynchronous notifications
Rbufs

- Proposed in the Nemesis microkernel-based multimedia OS
User-level drivers in a monolithic OS

Ben Leslie et al. User-level device drivers: Achieved performance, 2005

Diagram:
- User land
  - Driver
  - Linux Kernel
  - TCP/IP
- Application
User-level drivers in a monolithic OS

Ben Leslie et al. User-level device drivers: Achieved performance, 2005

Diagram:

- User land
- Mem-mapped I/O
- Linux Kernel
- Driver
- Application
- TCP/IP
- read()
- send()
User-level drivers in a monolithic OS

Ben Leslie et al. User-level device drivers: Achieved performance, 2005
Ben Leslie et al. User-level device drivers: Achieved performance, 2005

- **Performance**
  - Up to 7% throughput degradation
  - Up to 17% CPU overhead
  - Aggressive use of interrupt rate limiting potentially affects latency (not measured).
Nooks

- A complete device-driver reliability solution for Linux:
  - Fault isolation
  - Fault detection
  - Recovery
Nooks

- A complete device-driver reliability solution for Linux:
  - Fault isolation
  - Fault detection
  - Recovery

![Diagram showing the structure of Nooks with the Linux kernel, isolation manager, shadow driver, and driver components.]
Nooks

- A complete device-driver reliability solution for Linux:
  - Fault isolation
  - Fault detection
  - Recovery
Nooks

• A complete device-driver reliability solution for Linux:
  - Fault isolation
  - Fault detection
  - Recovery

• Problems
  - The driver interface in Linux is not well defined. Nooks must simulate the behaviour of hundreds of kernel and driver entry points.

• Performance
  - 10% throughput degradation
  - 80% CPU overhead
Virtualisation and user-level drivers

- Direct I/O
• Paravirtualised I/O
Paravirtualised I/O in Xen

- Xen I/O channels are similar to rbufs, but use a single circular buffer for both requests and completions and rely on mapping rather than sharing.
Xen I/O channels

Shared:
- req_prod
- req_event
- rsp_prod
- rsp_event

PUSH_REQUESTS updates this pointer
req_prod<--req_prod_pvt

Response producer:
- rsp_prod_pvt
- req_cons
- nr_ents
- *sring

Unconsumed requests:
0 1

255
Unconsumed responses

254

PUSH_RESPONSES updates this pointer
rsp_prod<--rsp_prod_pvt

Request producer:
- req_prod_pvt
- rsp_cons
- nr_ents
- *sring

Unconsumed responses

Paravirtualised I/O in Xen

• Performance overhead of the original implementation: 300%
  - Longer critical path (increased instructions per packet)
  - Higher TLB and cache miss rates (more cycles per instructions)
  - Overhead of mapping

• Optimisations
  - Avoid mapping on the send path (the driver does not need to “see” the packet content)
  - Replace mapping with copying on the receive path
  - Avoid unaligned copies
  - Optimised implementation of page mapping
  - CPU overhead down to 97% (worst-case receive path)
Other driver reliability techniques

• Implementing drivers using safe languages
  – Java OSs: KaffeOS, JX
    • Every process runs in a separate protection domain with a private heap. Process boundaries are enforced by the language runtime. Communication is based on shared heaps.
  – House (Haskell OS)
    • Bare-metal Haskell runtime. The kernel and drivers are in Haskell.
    • User programs can be written in any language.
  – SafeDrive
    • Extends C with pointer type annotations enforced via static and runtime checking
    • unsigned n;
      struct e1000 buffer * count(n) bufinfo;
Other driver reliability techniques

• Implementing drivers using safe languages
  - Singularity OS
    • The entire OS is implemented in Sing#
    • Every driver is encapsulated in a separate software-isolated process
    • Processes communicated via messages sent across channels
    • Sing# provides means to specify and statically enforce channel protocols
Other driver reliability techniques

- Static analysis
  - SLAM, Blast, Coverity
  - Generic programming faults
    - Release acquired locks; do not acquire a lock twice
    - Do not dereference user pointers
    - Check potentially NULL-pointers returned from routine
  - Driver-specific properties
    - “if a driver calls another driver that is lower in the stack, then the dispatch routine returns the same status that was returned by the lower driver”
    - “drivers mark I/O request packets as pending while queuing them”
  - Limitations
    - Many properties are beyond reach of current tools or are theoretically undecidable (e.g., memory safety)
Questions?
Part 3: Device drivers research at ERTOS
User-level device drivers

- What is the overhead of user-level I/O in a microkernel-based OS?
  - Still an open question
  - Indirect evidence suggest that the overhead can be reduced to ~10%

- Project: design, implement and evaluate a user-level driver framework for a modern microkernel (seL4 or OKL4)
Dingo: Taming Device Drivers
Leonid Ryzhyk    Peter Chubb    Ihor Kuz    Gernot Heiser
UNSW, NICTA, Open Kernel Labs
Can we develop drivers that contain fewer bugs in the first place?

Localise complexity in driver development

- Many driver bugs are provoked by the complexity of the OS interface

Reduce bugs by improving the design of this interface
Dingo for Linux

Dingo runtime

Dingo drivers

Native Linux drivers
A study of driver bugs
Driver defects

• Types of driver defects
  - Device protocol violations
  - OS protocol violations
  - Concurrency defects
  - Generic programming defects
## A study of Linux driver bugs

<table>
<thead>
<tr>
<th>Driver</th>
<th>#loc</th>
<th>#bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>USB</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTL8150 USB-to-Ethernet adapter</td>
<td>827</td>
<td>16</td>
</tr>
<tr>
<td>EL1210a USB-to-Ethernet adapter</td>
<td>710</td>
<td>2</td>
</tr>
<tr>
<td>KL5kusb101 USB-to-Ethernet adapter</td>
<td>925</td>
<td>15</td>
</tr>
<tr>
<td>Generic USB network driver</td>
<td>1028</td>
<td>45</td>
</tr>
<tr>
<td>USB hub</td>
<td>2234</td>
<td>67</td>
</tr>
<tr>
<td>USB-to(serial converter)</td>
<td>989</td>
<td>50</td>
</tr>
<tr>
<td>USB mass storage</td>
<td>803</td>
<td>23</td>
</tr>
<tr>
<td><strong>Firewire</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IEEE1394 Ethernet controller</td>
<td>1413</td>
<td>22</td>
</tr>
<tr>
<td>SBP-2 transport protocol</td>
<td>1713</td>
<td>46</td>
</tr>
<tr>
<td><strong>PCI</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mellanox InfiniHost InfiniBand adapter</td>
<td>11718</td>
<td>123</td>
</tr>
<tr>
<td>BNX2 Ethernet adapter</td>
<td>5412</td>
<td>51</td>
</tr>
<tr>
<td>i810 frame buffer</td>
<td>2920</td>
<td>16</td>
</tr>
<tr>
<td>CMI8338 audio</td>
<td>2660</td>
<td>22</td>
</tr>
</tbody>
</table>

Total: 498
A study of Linux driver bugs

OS protocol

Driver

device protocol
A study of Linux driver bugs

Device protocol violation examples:
- Issuing a command to uninitialised device
- Writing an invalid register value
- Incorrectly managing DMA descriptors
Device protocol violations

38%
if(cur_state==IB_RESET && new_state==IB_RESET){
    return 0;
}
OS protocol violations

- Device protocol violations: 38%
- OS protocol violations: 20%
Concurrency errors

- Race in config functions: 30
- Race in hot unplug handler: 25
- Deadlock in an atomic context: 20
- Race in the data path: 15
- Race in PM functions: 10
- Uninitialised lock: 5
- Imbalanced locks: 5
- Other: 0
Concurrency errors

- Race in config functions:
- Race in hot unplug handler:
- Deadlock in an atomic context:
- Race in the data path:
- Race in PM functions:
- Uninitialised lock:
- Imbalanced locks:
- Other:
Concurrency errors

- Race in config functions:
- Race in hot unplug handler:
- Deadlock in an atomic context:
- Race in the data path:
- Race in PM functions:
- Uninitialised lock:
- Imbalanced locks:
- Other:
Concurrency errors

- Device protocol violations: 38%
- OS protocol violations: 19%
- Concurrency errors: 20%
Generic errors

- Device protocol violations: 38%
- OS protocol violations: 19%
- Concurrency errors: 23%
- Generic errors: 20%
Dealing with concurrency bugs
Dealing with concurrency bugs

Threads

request1

request2

irq
Dealing with concurrency bugs

Threads

Events

request1

request2

irq

request1

request2

irq

Dingo

evt3

evt2

evt1

driver
int probe ()
{
    ...
    write_config_reg ();
    msleep(20);
    read_status_reg ();
    ...
}

void probe ()
{
    ...
    write_config_reg ();
    timeout(20, probe2);
}

void probe2 ()
{
    read_status_reg ();
    ...
}
int probe ()
{
    ...
    write_config_reg ();
    msleep(20);
    read_status_reg ();
    ...
}

void probe ()
{
    simple_evt notif;
    ...
    write_config_reg ();
    CALL (timeout(20), notif);
    read_status_reg ();
    ...
}
Performance of the AX88772 USB-to-Ethernet adapter driver

Evaluation platform: 4 x 2GHz Itanium II (SMT, 2 threads per core)

- CPU Utilisation (%)
  - Blue: Linux
  - Red: Dingo

- Round-Trip (μsec)
  - X-axis: Number of Connections
  - Y-axis: Round-Trip (μsec)
Impact of serialisation on performance

Special case: drivers for very-high-performance devices

- Examples: 10Gb Ethernet, Infiniband
- For such drivers, serialisation affects performance on multiprocessors

Solution: Re-introduce multithreading at the data path

- Avoid concurrency bugs at the control path, while maintaining high performance at the data path
Performance of the Mellanox InfiniBand adapter driver

- **CPU Utilisation (%)**
  - Number of Connections: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32
  - Throughput (Mb/s): 0, 10, 20, 30, 40, 50

- Lines:
  - **Linux**
  - **Dingo (serialised)**
  - **Dingo (multithreaded)**

NICTA
Dealing with OS protocol violations
Modeling driver protocols with state machines

? - incoming call from the OS
! - outgoing call to the OS

init

start

?start

!startComplete

running

?stop

stop

!stopComplete

unplugged

?unplugged

!stopComplete
Other features of the specification language:

- Timeouts
- Protocol variables
- Dynamic protocol spawning
- etc.
Runtime failure detection

Driver

OS protocol
Runtime failure detection

EthernetController protocol SM

Driver

OS protocol
Current status

• Current status of Dingo
  - Building an open-source implementation of the Dingo architecture in Linux

• Project
  - Implement and evaluate device drivers for the Dingo architecture
Automatic Device Driver Synthesis with Termite

Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, Gernot Heiser

UNSW, NICTA, Open Kernel Labs
Device drivers today

Device datasheet

OS documentation/headers

Driver implementation
Device drivers today

- Incomplete, inaccurate
  - Device datasheet
- Incomplete
  - (use the source, Luke)
  - OS documentation/
    headers

Driver implementation

```c
if (request_irq(i8042_ports[I8042_AUX_PORT_NO].irq, i8042_interrupt,
    SA_SHIRQ, "i8042", &i8042_check_aux_cookie))
    return -1;
free_irq(i8042_ports[I8042_AUX_PORT_NO].irq, &i8042_check_aux_cookie);
```
Drivers have more errors/LOC than any other OS component (by an OOM)

Driver implementation
Driver synthesis: a high-level view

Formal device protocol specification

Formal driver/OS protocol specification

Driver implementation

```c
if (request_irq(i8042_ports[I8042_AUX_PORT_NO].irq, i8042_interrupt, 
    SA_SHIRQ, "i8042", &i8042_check_aux_cookie))
    return -1;
free_irq(i8042_ports[I8042_AUX_PORT_NO].irq, &i8042_check_aux_cookie);
```
Driver synthesis: a high-level view

Formal device protocol specification

Formal driver/OS protocol specification

Driver implementation
dummy-net device

<table>
<thead>
<tr>
<th>ctrl</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0=off</td>
<td></td>
</tr>
<tr>
<td>1=on</td>
<td></td>
</tr>
</tbody>
</table>
Driver synthesis by example

OS protocol specification:

```
ctrl:=1
ctrl:=0
data:=1/
sent
!sendComplete
```

dummy-net device

<table>
<thead>
<tr>
<th>ctrl</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0=off</td>
<td></td>
</tr>
<tr>
<td>1=on</td>
<td></td>
</tr>
</tbody>
</table>

Device protocol specification:

```
ctrl:=1
ctrl:=0
data:=1/
sent
```
Driver synthesis by example

OS protocol specification:

dummy-net device

0=off
1=on

ctrl data

Device protocol specification:

dummy-net driver

ctrl:=1
data:=1/sent
ctrl:=0
Driver synthesis by example

OS protocol specification:

!sendComplete

?send

sent

dummy-net device

0=off
1=on

ctrl
data

dummy-net driver

Device protocol specification:

ctrl:=1

data:=1/
sent

ctrl:=0
Driver synthesis by example

OS protocol specification:

dummy-net device

<table>
<thead>
<tr>
<th>ctrl</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0=off</td>
<td></td>
</tr>
<tr>
<td>1=on</td>
<td></td>
</tr>
</tbody>
</table>

Device protocol specification:

dummy-net driver
Driver synthesis by example

OS protocol specification:

dummy-net device

<table>
<thead>
<tr>
<th>ctrl</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0=off</td>
<td></td>
</tr>
<tr>
<td>1=on</td>
<td></td>
</tr>
</tbody>
</table>

dummy-net driver

Device protocol specification:
Driver synthesis by example

OS protocol specification:

dummy-net device

<table>
<thead>
<tr>
<th>ctrl</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0=off</td>
<td></td>
</tr>
<tr>
<td>1=on</td>
<td></td>
</tr>
</tbody>
</table>

dummy-net driver

Device protocol specification:
Driver synthesis by example

OS protocol specification:

?send

sent

!sendComplete

dummy-net device

<table>
<thead>
<tr>
<th></th>
<th>ctrl</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>off</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>on</td>
<td></td>
</tr>
</tbody>
</table>

dummy-net driver

Device protocol specification:

ctrl:=1

data:=1/
sent

ctrl:=0
Driver synthesis by example
Driver synthesis by example
Driver synthesis by example

\[
\begin{align*}
?send & \quad \text{sent} \\
\text{!sendComplete} & \\
\text{!sendComplete} & \\
\text{ctrl} &= 1 \\
\text{ctrl} &= 0 \\
\text{data} &= 1/ \\
\text{!sendComplete} & \\
\end{align*}
\]
Driver synthesis by example

![Diagram of driver synthesis process]

- ?send
- sent
- !sendComplete
- ctrl:=1
- data:=1/
- !sendComplete
- !sendComplete
- ctrl:=1
- data:=1/
- !sendComplete
Driver synthesis by example
Driver synthesis by example
Driver synthesis by example
Driver synthesis by example

Diagram showing state transitions with labels such as `?send(1)`, `sent(1)`, `!sendComplete`, `ctrl:=1`, `data:=1/sent(1)`, and `!sendComplete`.
Implementation

ASIX 88772 100Mb/s USB-to-Eth controller driver
Questions?