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 38 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

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

IBM 7090 [1959] introduced I/O channels, which allowed I/O and computation to overlap

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.

---

**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**

---

**PCI bus overview: DMA**

---

**PCI bus overview: interrupts**
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

I/O devices in a typical desktop system

Driver stacking
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
  - 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-level drivers in \( \mu \)-kernel OSs

![Diagram of User-level drivers in \( \mu \)-kernel OSs](image)

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 \( \mu \)-kernel-based multi-server OS
  - High IPC overhead
  - Eventually, moved drivers back into the kernel
- L3 [1987-1993]
  - Persistent \( \mu \)-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 Raccoon-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 queuing
  – 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

- 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
Nooks

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

Virtualisation and user-level drivers

• Direct I/O

Paravirtualised I/O in Xen

Xen I/O channels

* 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
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;

- 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 routines
  - 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  Gerot Heiser
UNSW, NICTA, Open Kernel Labs

The Dingo approach

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

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>RTL8150 USB-to-Ethernet adapter</td>
<td>927</td>
<td>16</td>
</tr>
<tr>
<td>EL1210a USB-to-Ethernet adapter</td>
<td>710</td>
<td>2</td>
</tr>
<tr>
<td>KL5xusb101 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>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>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>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>498</strong></td>
</tr>
</tbody>
</table>

#### Device protocol violations

- Issuing a command to uninitialised device
- Writing an invalid register value
- Incorrectly managing DMA descriptors

#### OS protocol violations

```c
if(cur_state==IB_RESET &&
new_state==IB_RESET){
    return 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:

### Device protocol violations

- Deadlock in an atomic context:
- Race in the data path:
- Race in PM functions:
- Uninitialised lock:
- Imbalanced locks:
- Other:

### OS protocol violations

### Concurrency errors

### Generic errors

- Device protocol violations
- OS protocol violations
- Concurrency errors
- Generic errors

### Dealing with concurrency bugs
Dealing with concurrency bugs

Threads

Events

Dingo

Writing non-blocking drivers

Linux

Dingo

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

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

void probe2 ()
{
    read_status_reg ();
    ...
}

Dingo

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

void probe2 ()
{
    read_status_reg ();
    ...
}

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

Dealing with OS protocol violations

Modeling driver protocols with state machines

Ethernet controller protocol fragment

Other features of the language

Other features of the specification language:
- Timouts
- Protocol variables
- Dynamic protocol spawning
- etc.
Runtime failure detection

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

Drivers have more errors/LOC than any other OS component (by an OOM)
Driver synthesis: a high-level view

Formal device protocol specification

Formal driver/OS protocol specification

Driver implementation

Driver synthesis by example

dummy-net device

0=off 1=on

ctrl data

OS protocol specification:

7send send

tsendComplete

dummy-net device

0=off 1=on

ctrl data

Device protocol specification:

cst=1

data=1/ sent

cnt=0

Driver synthesis by example

dummy-net device

0=off 1=on

ctrl data

OS protocol specification:

7send send

tsendComplete

dummy-net device

0=off 1=on

ctrl data

Device protocol specification:

cst=1

data=1/ sent

cnt=0
ASIX 88772 100Mbit USB-to-Eth controller driver

Questions?