Device Drivers

COMP9242

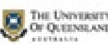
2009/S2 Week 9

Australian Government

Department of Broadband, Communications and the Digital Economy

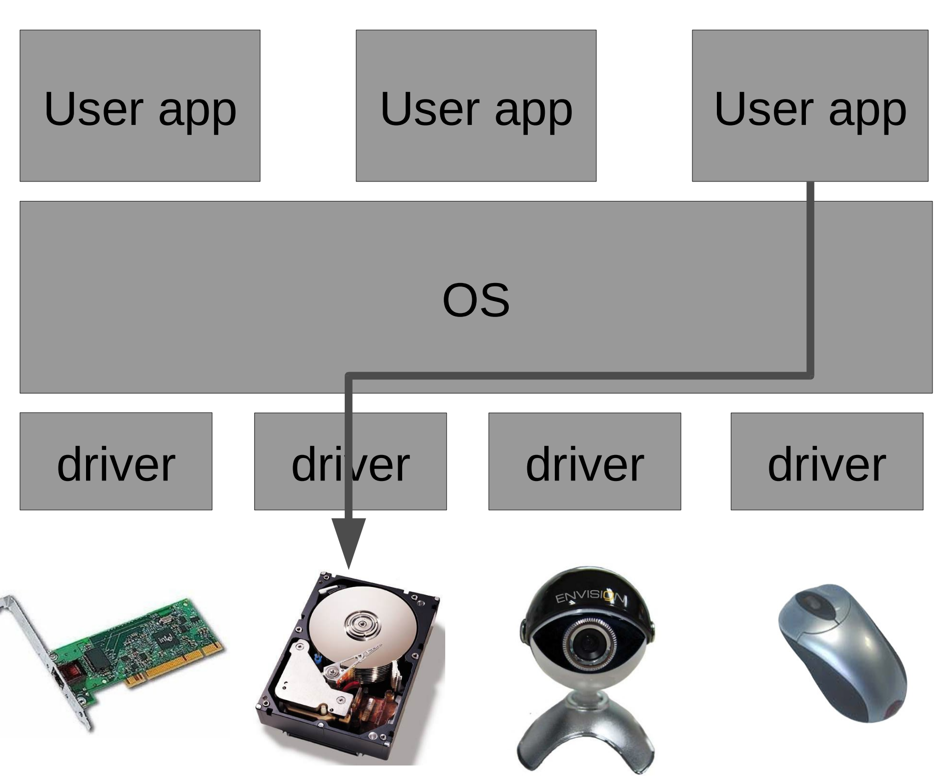
Australian Research Council

NICTA Members



NICTA Partners

NICTA


User app
User app
User app
OS
driver
driver
driver
driver
ENVISION

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

OS archeology

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

OS archeology

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."

Robert F. Rosin. Supervisory and monitor systems. ACM Computing Surveys, 1(1):37–54, 1969.

OS archeology

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

OS archeology

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

I/O adapter programs

Interface I:
character devices

Interface II:
block devices

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)

CPU

Dev1 Dev2 Dev3

PCI controller

RAM

PCI memory space

Dev1

Dev2

Dev3

PCI bus overview: DMA

Physical address space (FSB)

CPU

Dev1 Dev2 Dev3

PCI controller

RAM

PCI memory space

Dev1

Dev2

Dev3

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

USB bus overview

Device 3
Device 4
Device 1
Device 2
Hub
Root hub
DMA
USB bus controller
DMA
Transfer descriptors
Completions

I/O devices in a typical desktop system

Memory controller

Interrupt controller

CPU

Front-side bus

Ethernet controller (BCM4401)

PCI bus controller (ICH7)

PCI bus

Graphics controller (GMA950)

USB bus

USB controller (EHCI)

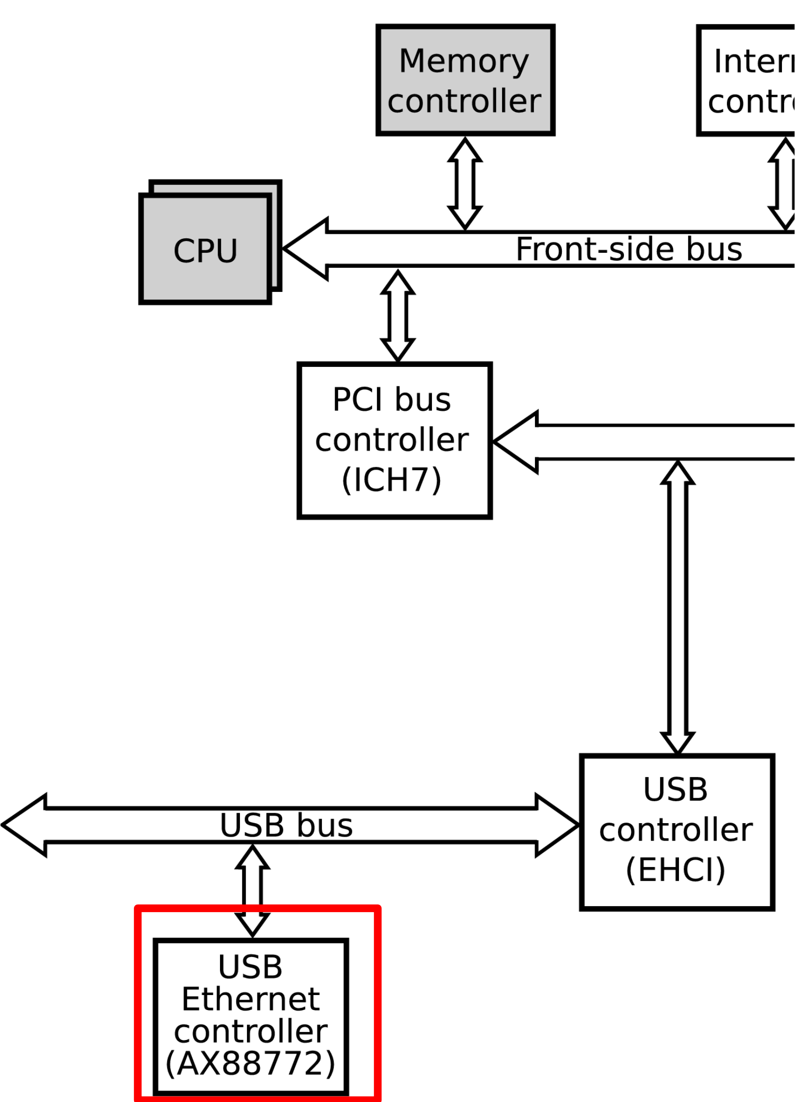
SATA bus controller (AHCI)

SATA bus

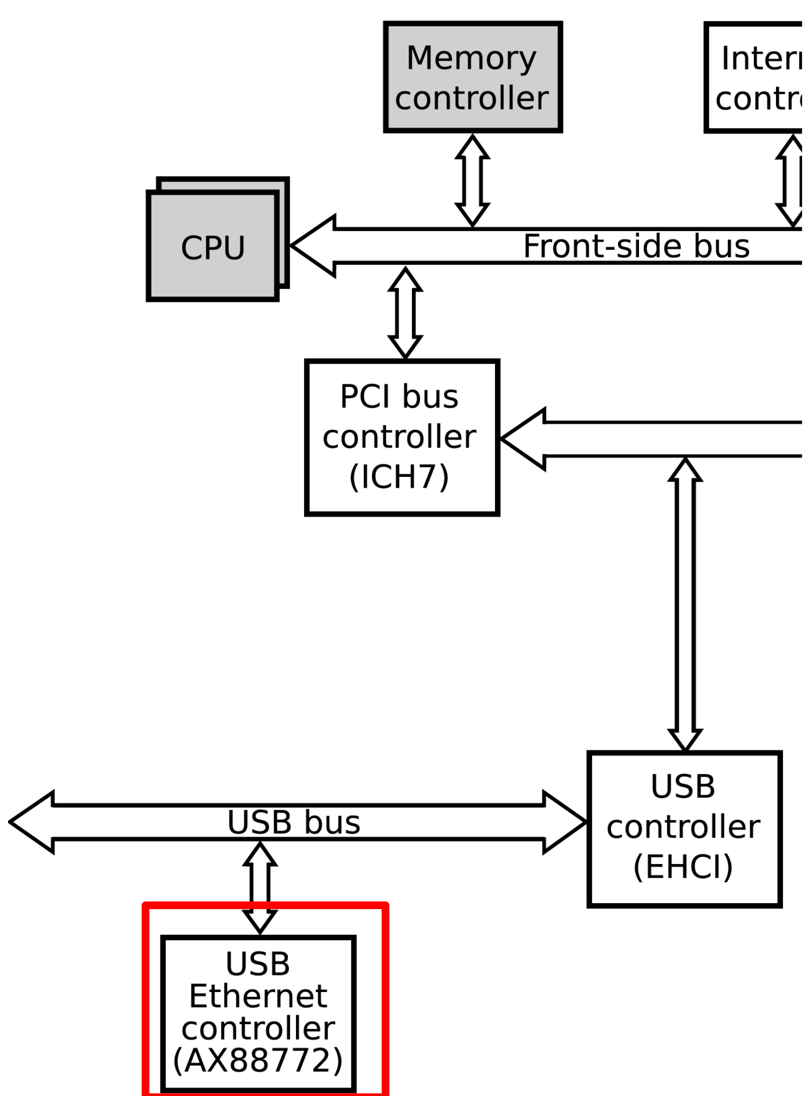
USB Ethernet controller (AX88772)

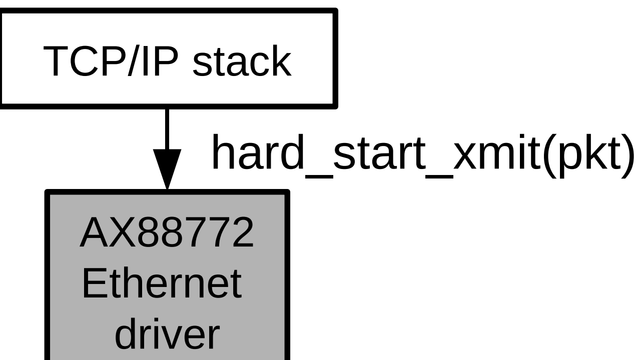
SATA disk

Driver stacking

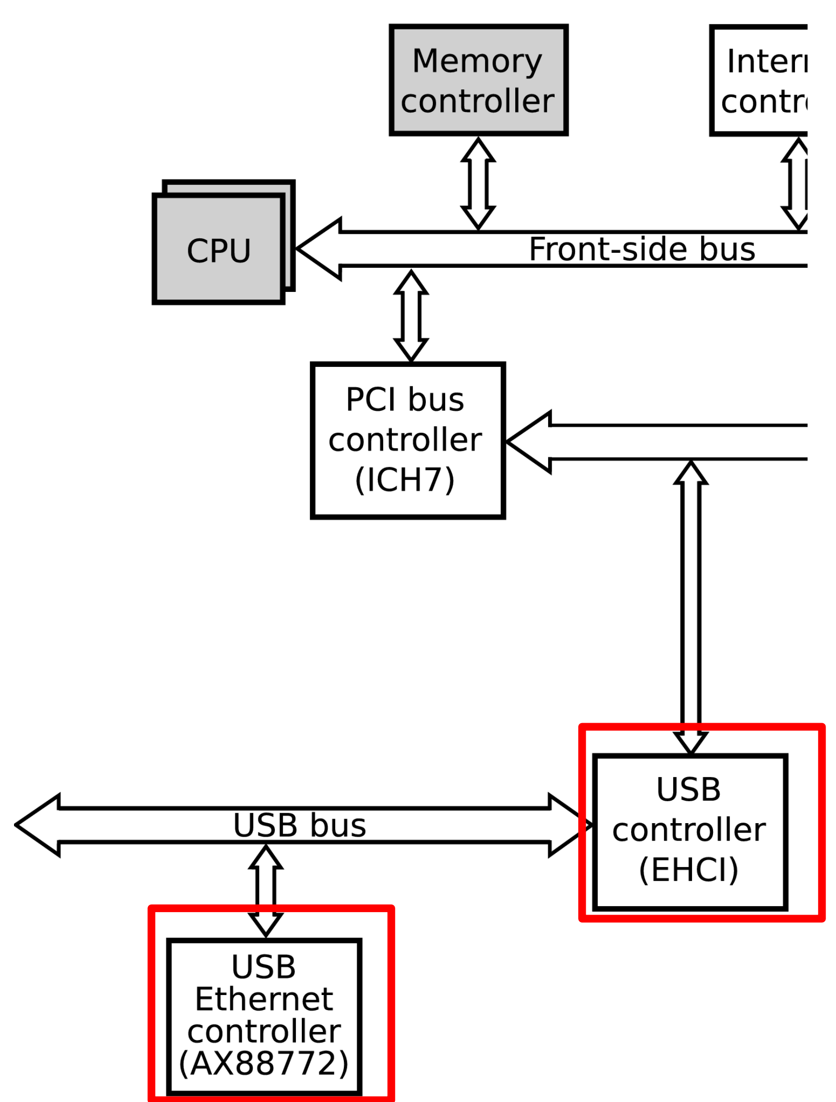


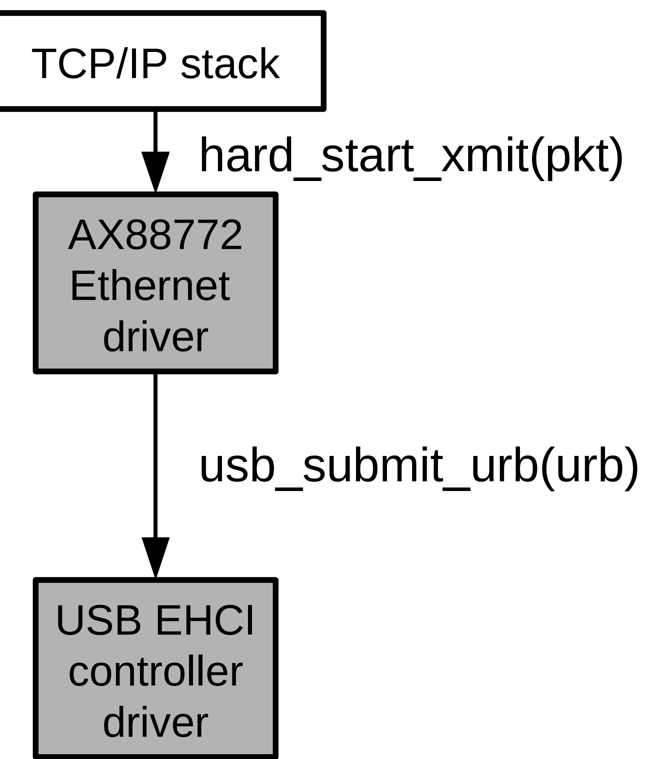
Driver stacking



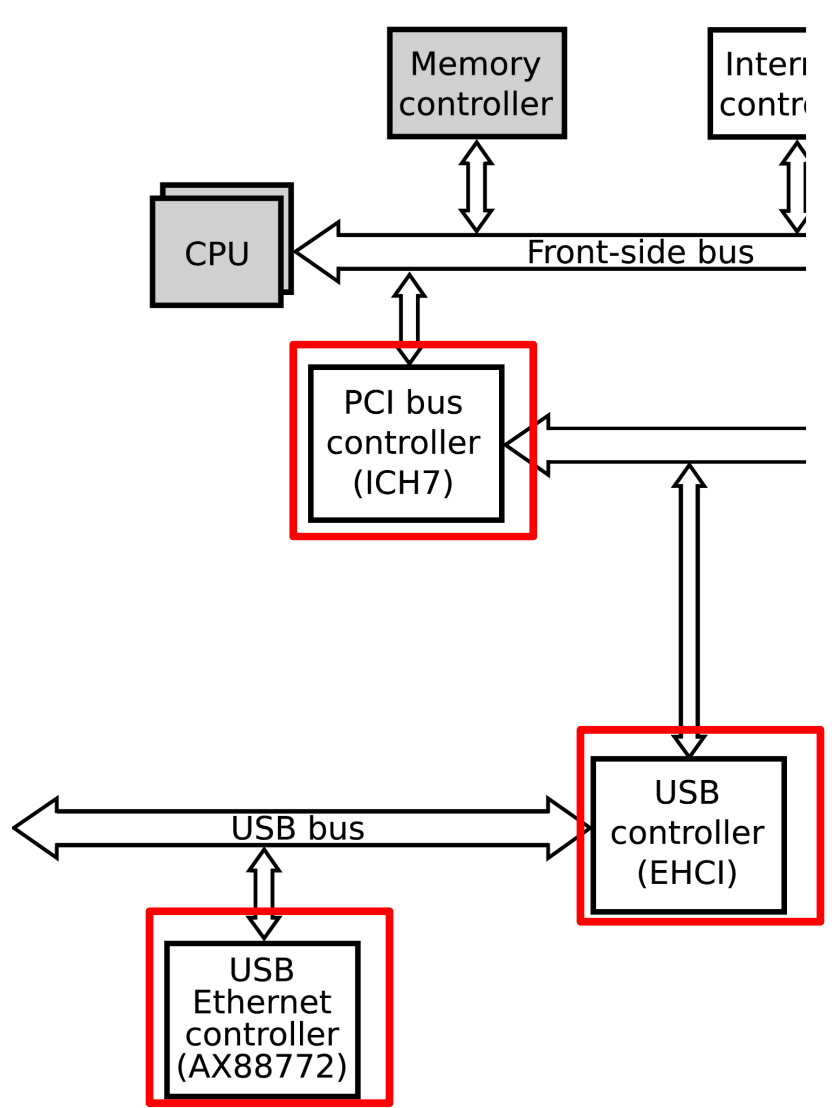


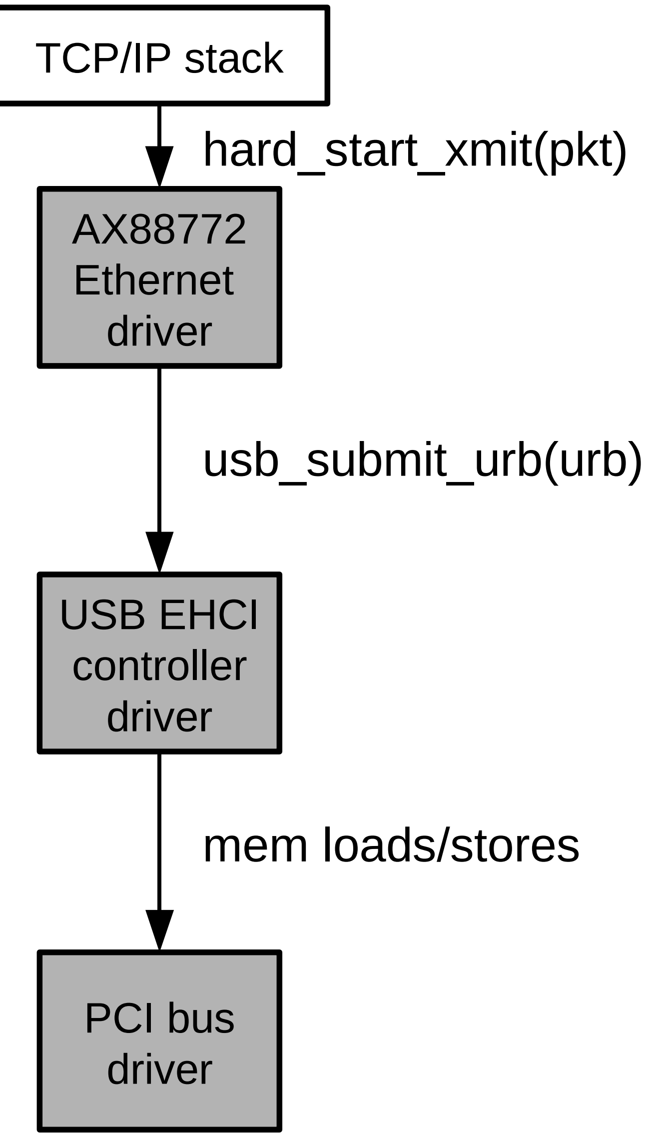
Driver stacking



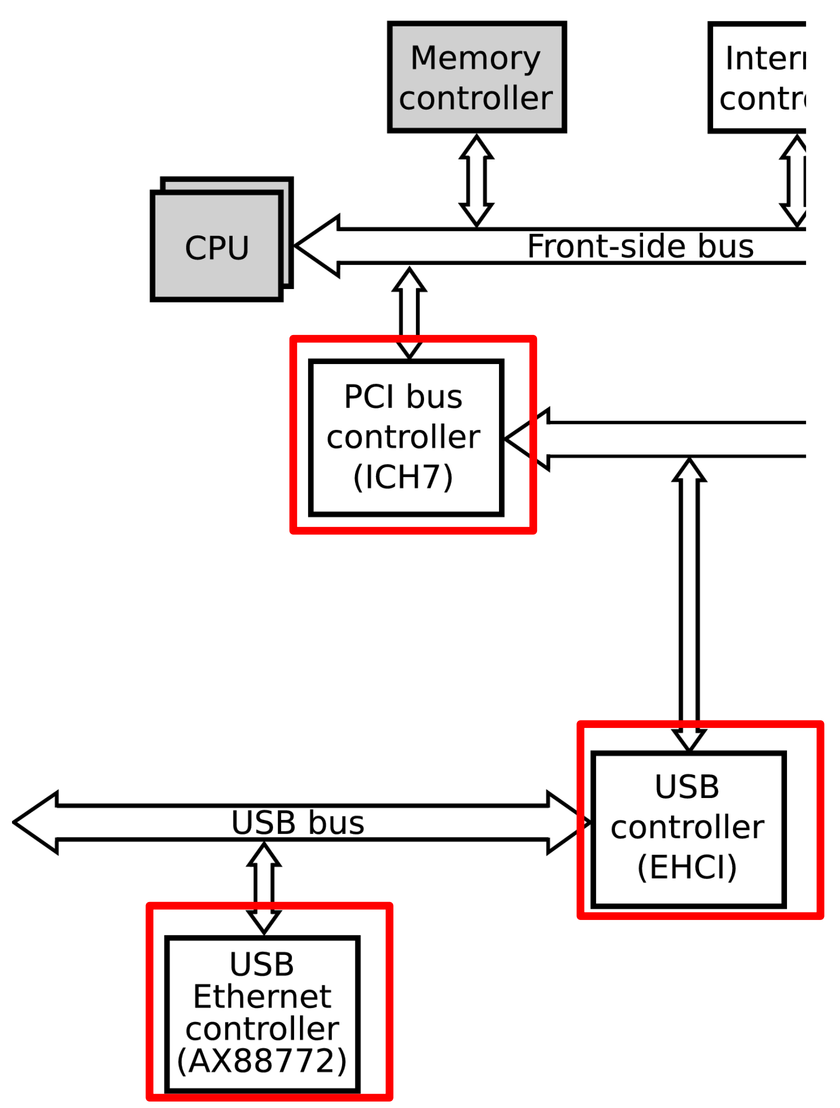


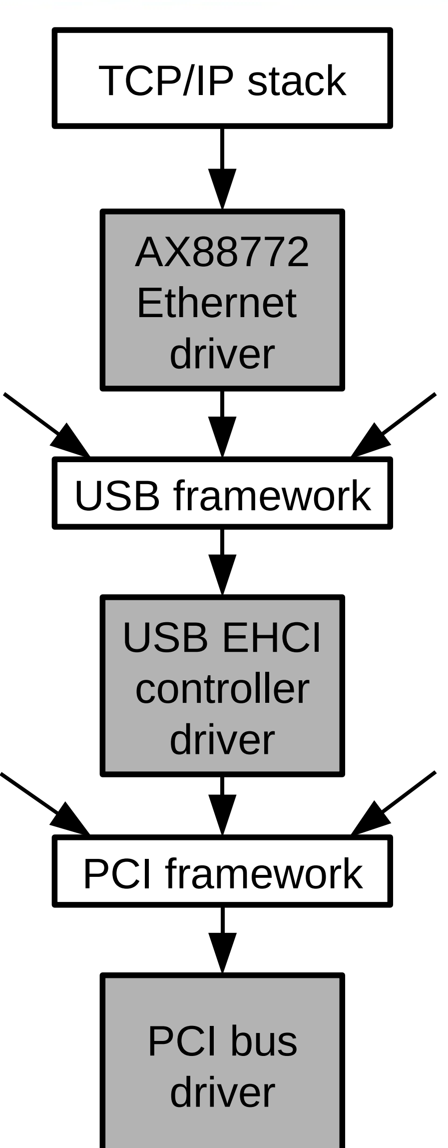
Driver stacking





Driver stacking





Driver framework design patterns

Client

Device-class interface

Driver

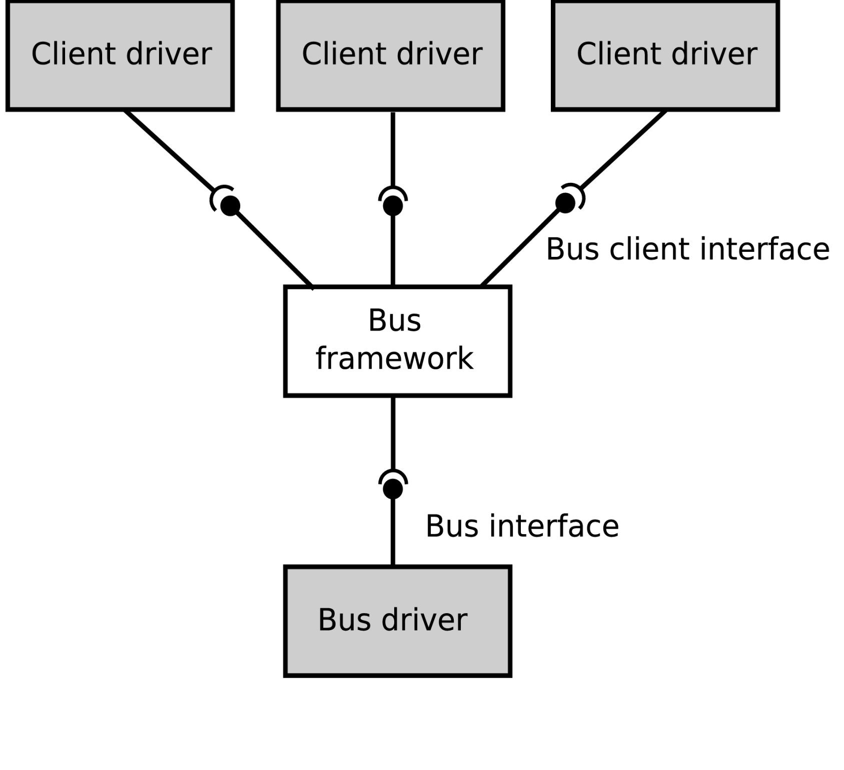
Bus client interface

Bus framework

Device protocol

Device

The driver pattern



The bus pattern

Driver framework software architecture

TCP/IP stack
File system
Window system

I/O framework

Block layer

AX88772 Ethernet driver
Generic SCSI disk driver

USB framework
ATA framework

BCM4401 Ethernet driver
USB EHCI controller driver
AHCI controller driver
GMA950 controller driver

PCI framework

Generic PCI 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

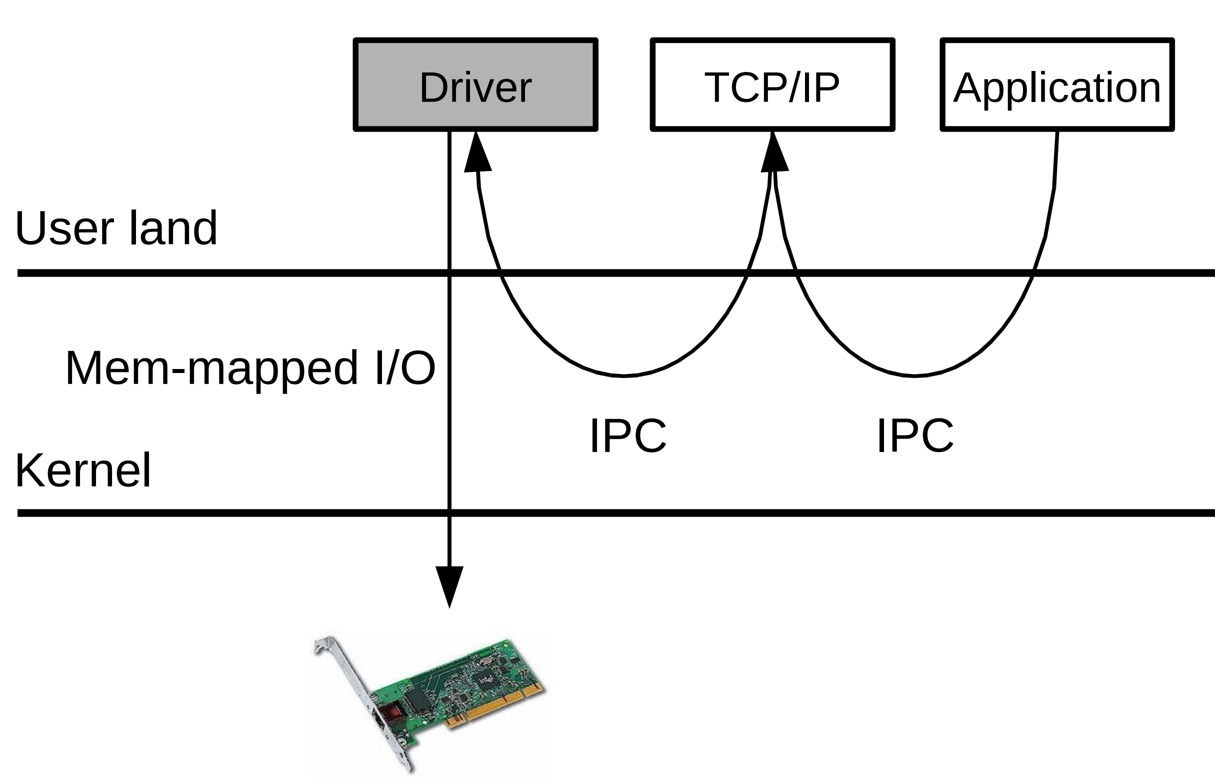
Understanding driver bugs

- 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

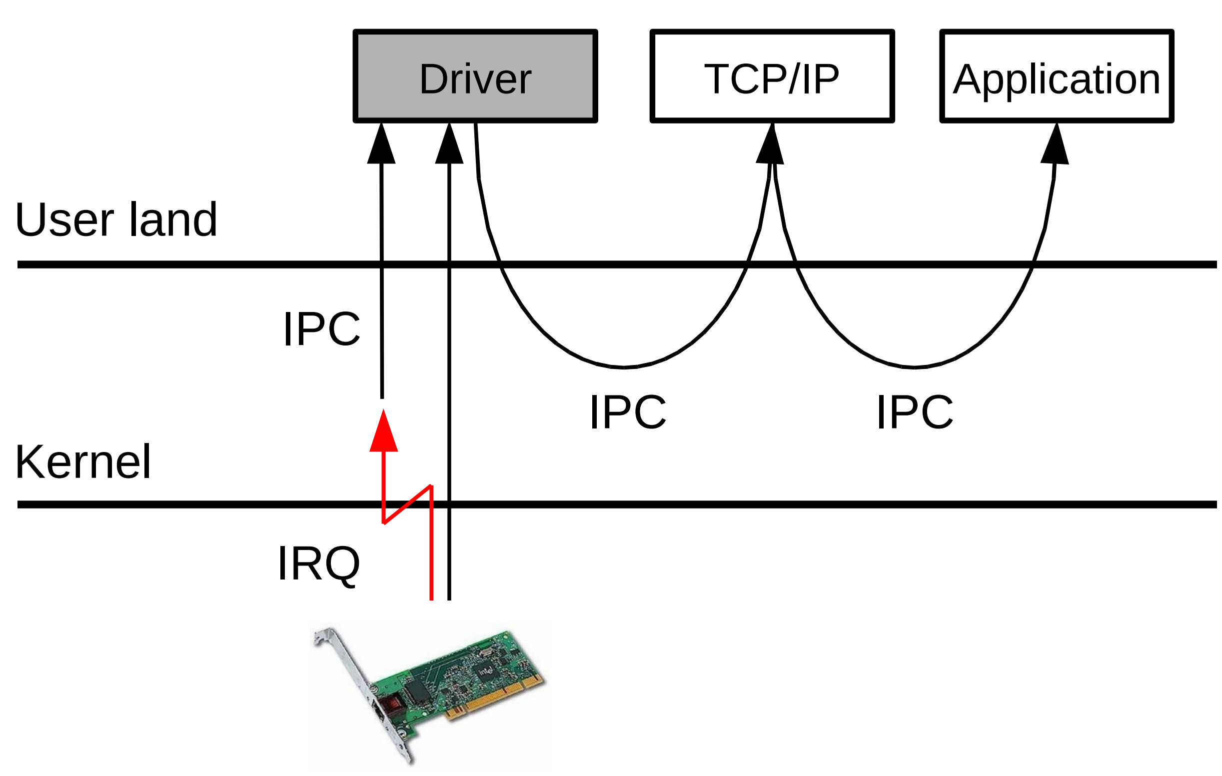
User-level device drivers

- 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-level drivers in µ-kernel OSs



User-level drivers in µ-kernel OSs

Net filter

Driver

Application

TCP/IP

User land

IPC

Kernel

User-level drivers in µ-kernel OSa

TCP/IP

Driver

Application

User land

IPC

Kernel

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

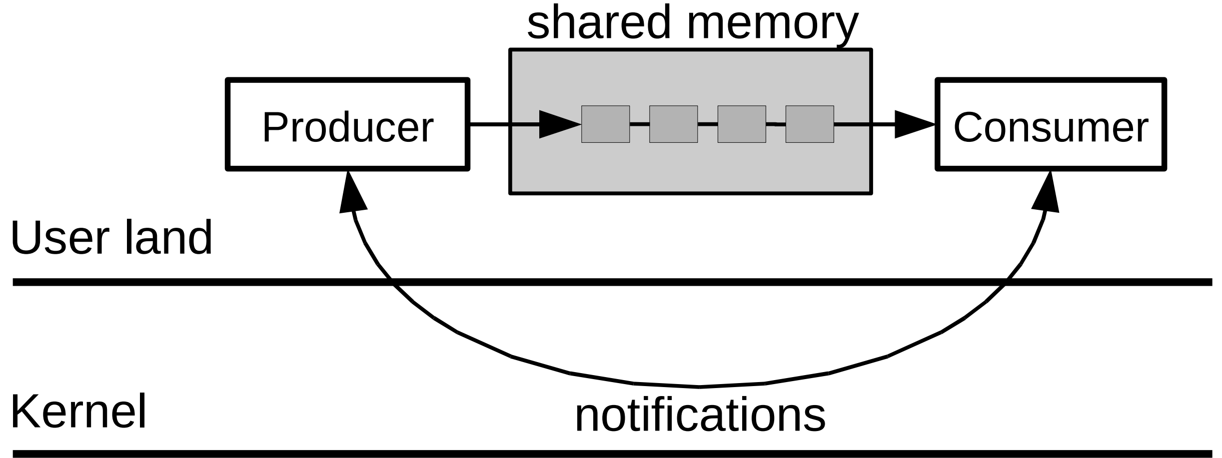
- Research
  - seL4
  - MINIX3
  - Nexus
- Commercial
  - OKL4
  - QNX
  - GreenHills INTEGRITY

Improving the performance of ULD

Improving the performance of ULD

- Ways to improve user-level driver performance
  - Shared-memory communication
  - Request queueing
  - Interrupt coalescing

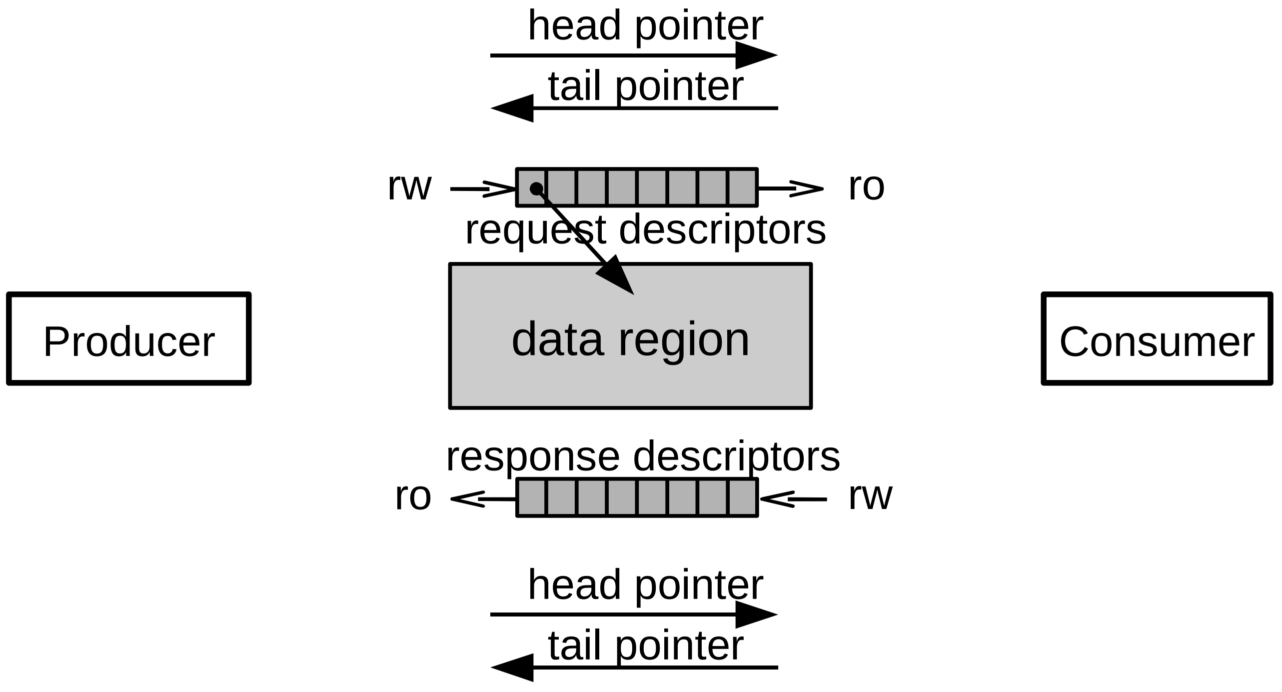
Implementing efficient shared-memory communication



- 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

Driver

Application

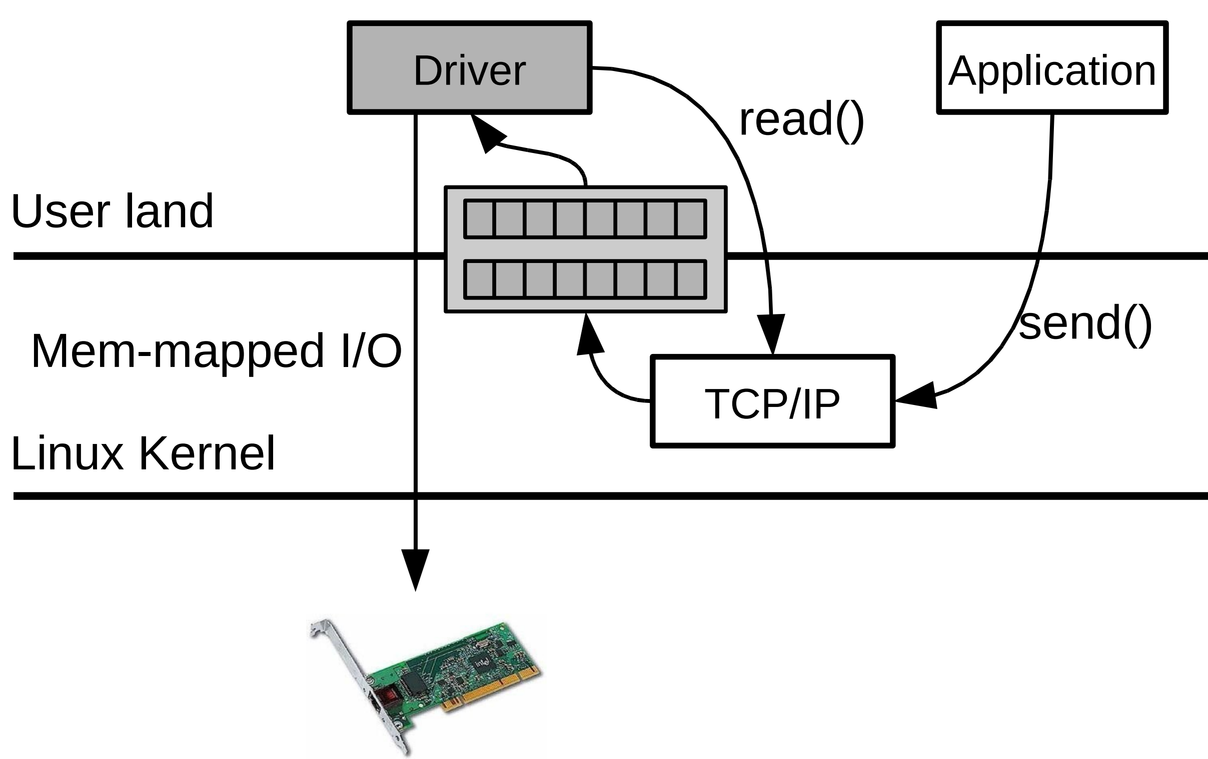
User land

TCP/IP

Linux Kernel

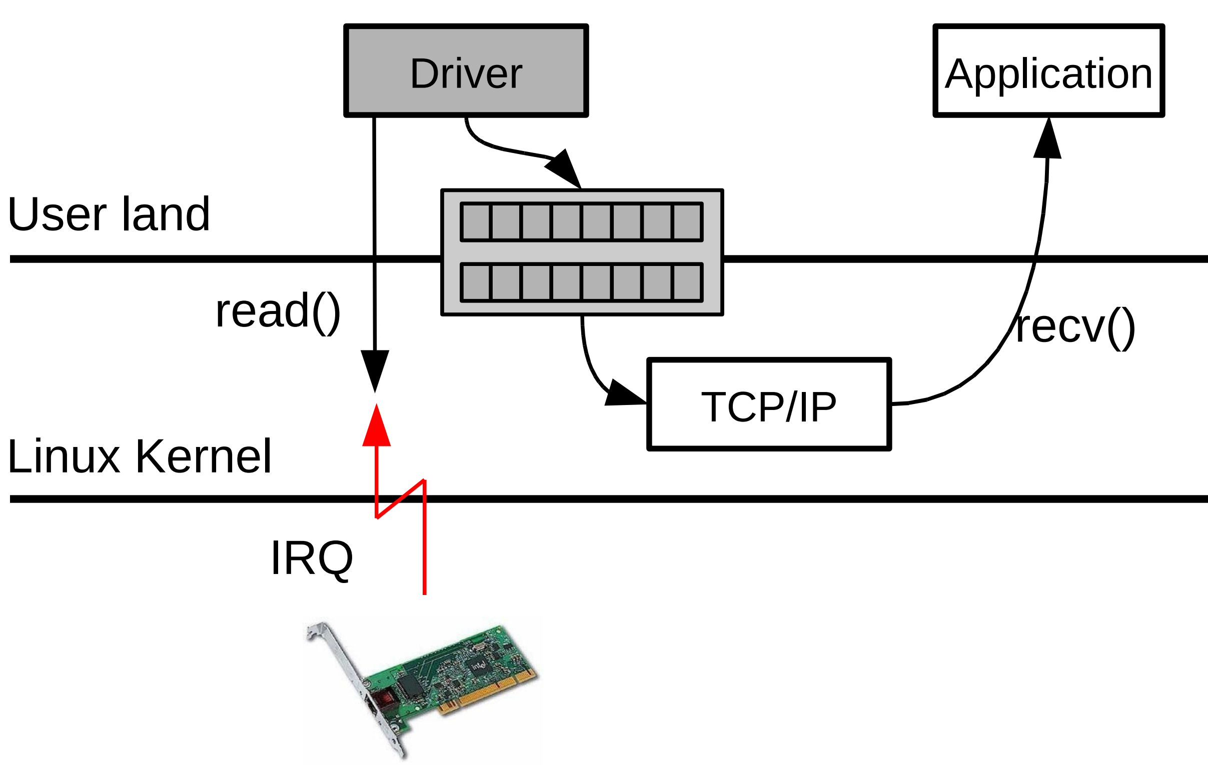
User-level drivers in a monolithic OS

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



User-level drivers in a monolithic OS

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



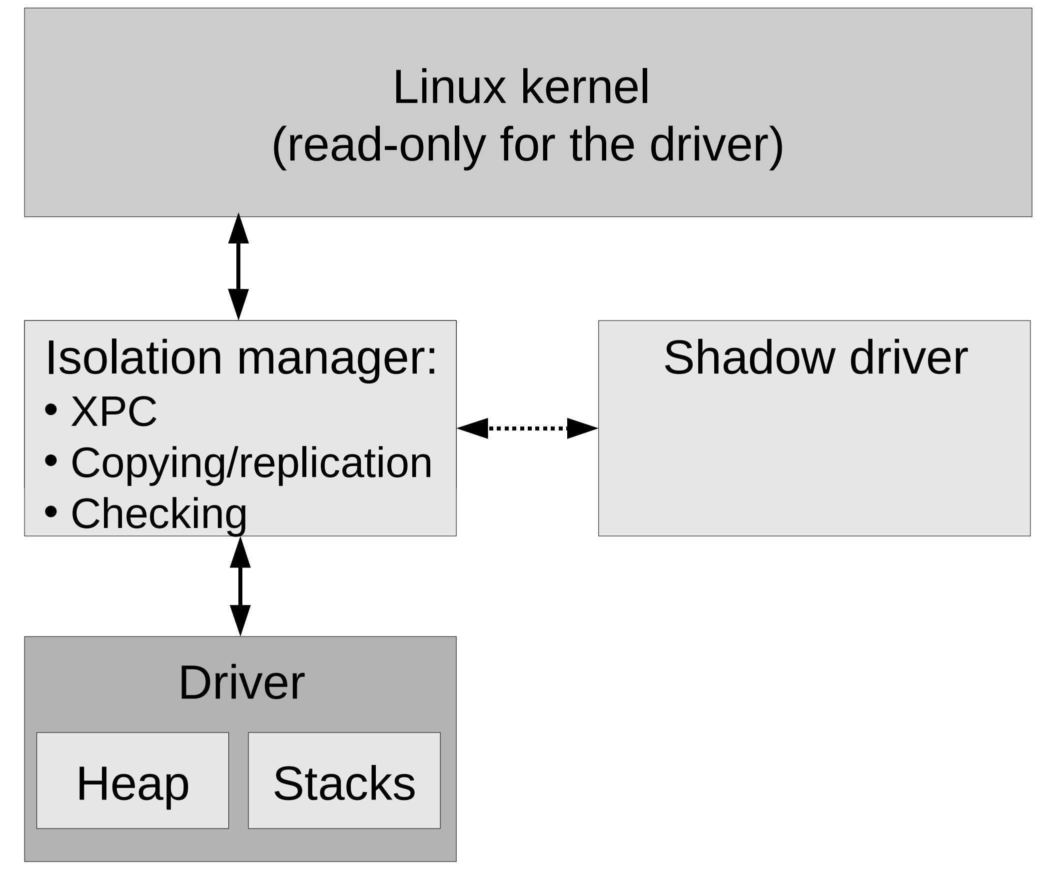
User-level drivers in a monolithic OS

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).

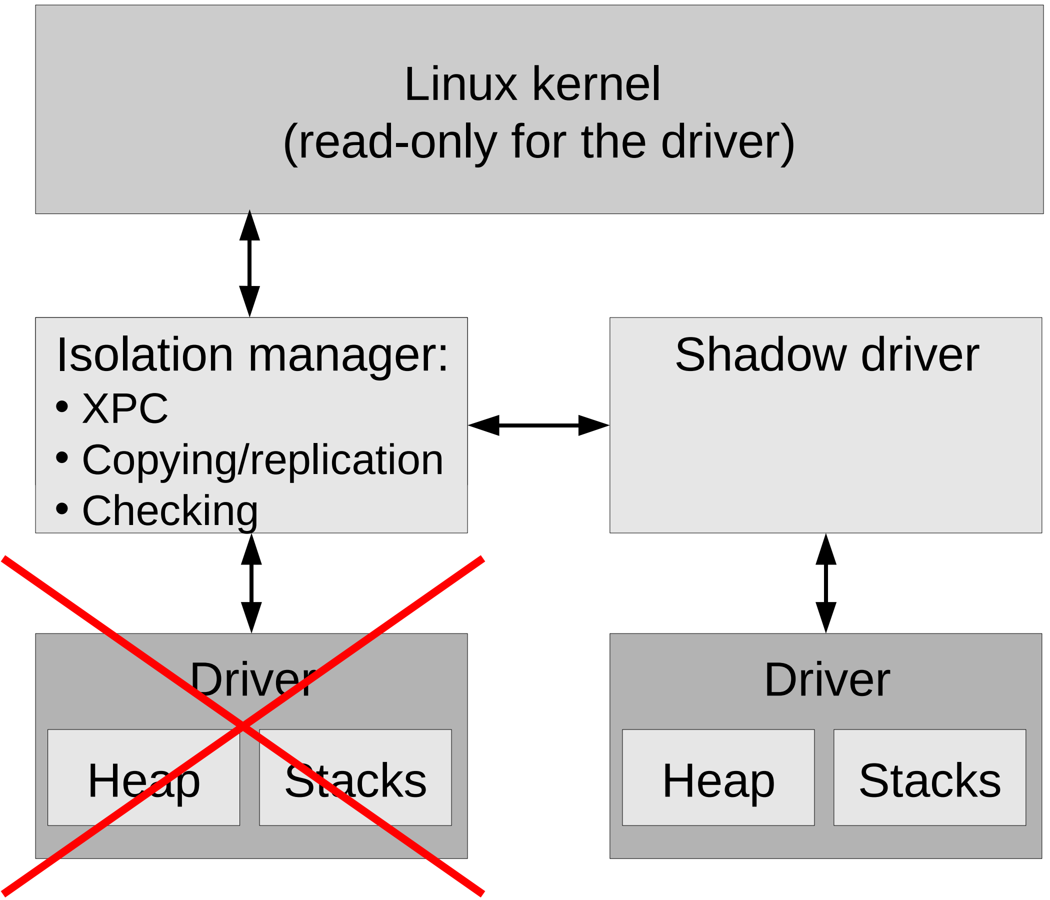
Nooks

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



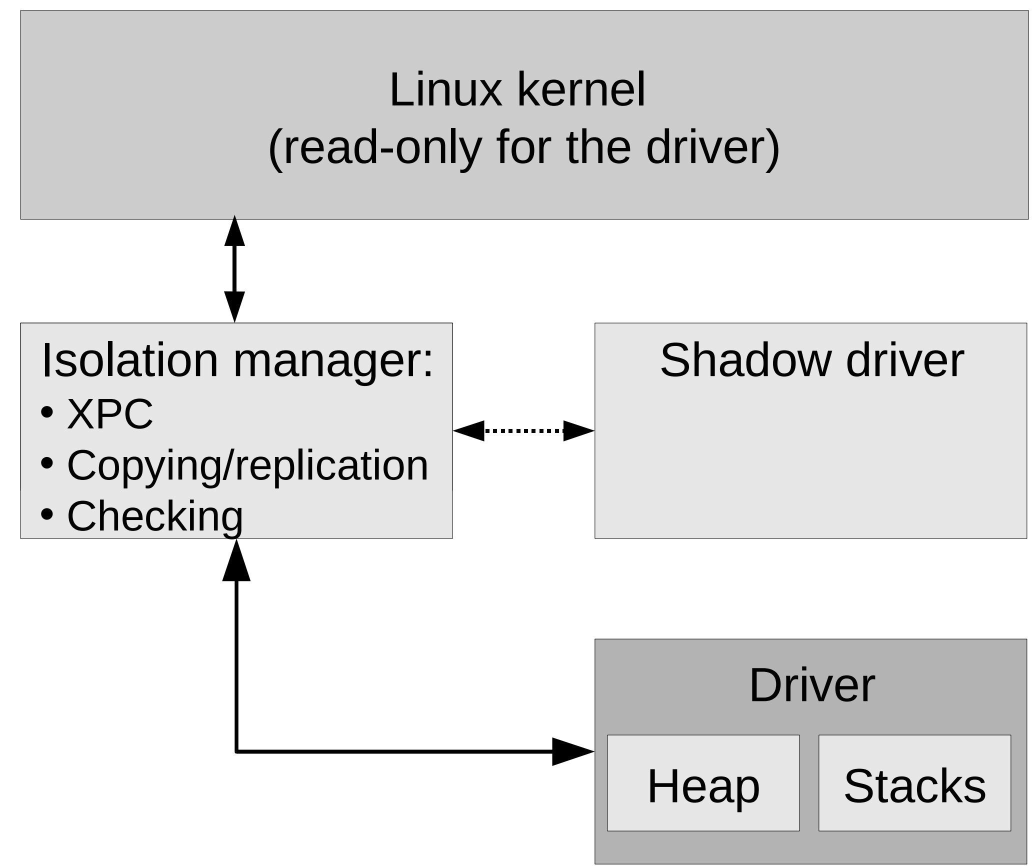
Nooks

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



Nooks

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

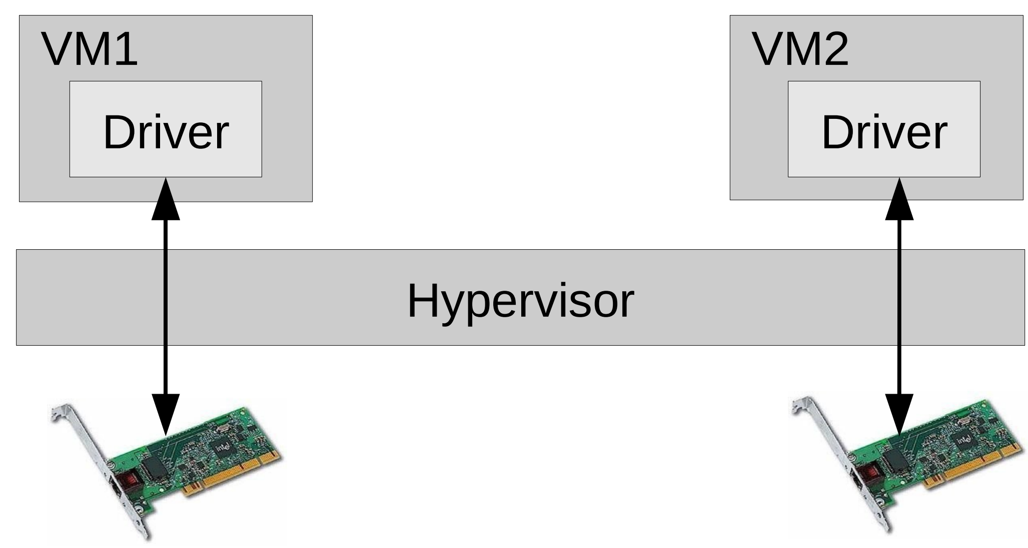


Nooks

- 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

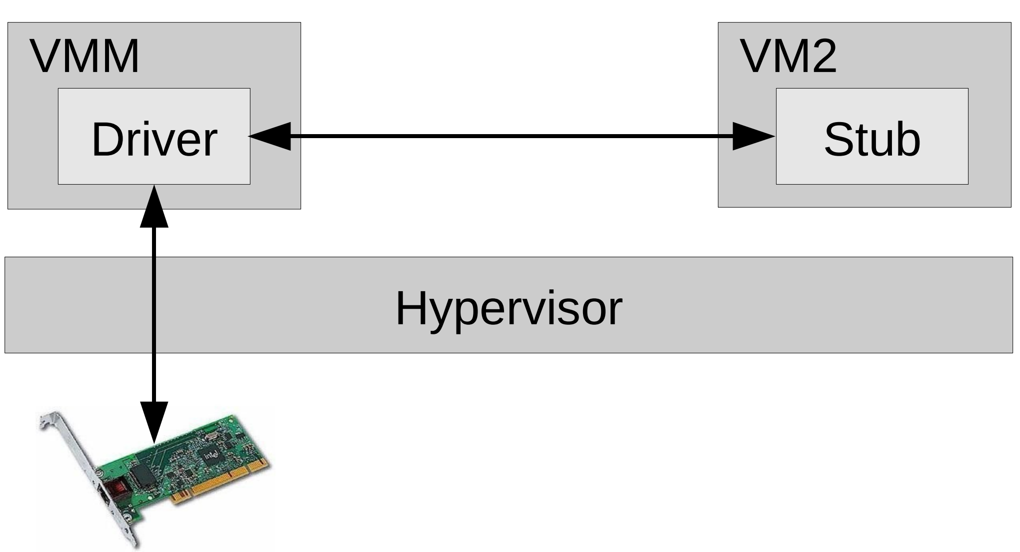
Virtualisation and user-level drivers

- Direct I/O



Virtualisation and user-level drivers

- Paravirtualised I/O



Paravirtualised I/O in Xen

Driver domain

netback

free buf

rx packet

Driver

I/O channels

TX

RX

Guest domain

netfront

tx buf

rx buf

Xen hypervisor

- 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

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

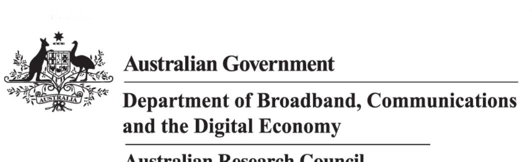
User-level device drivers

- 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)

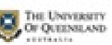
Dingo: Taming Device Drivers

Leonid Ryzhyk Peter Chubb Ihor Kuz Gernot Heiser

UNSW, NICTA, Open Kernel Labs



NICTA Members



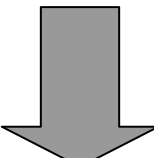
NICTA Partners

The Dingo approach

Can we develop drivers that contain fewer bugs in the first place?

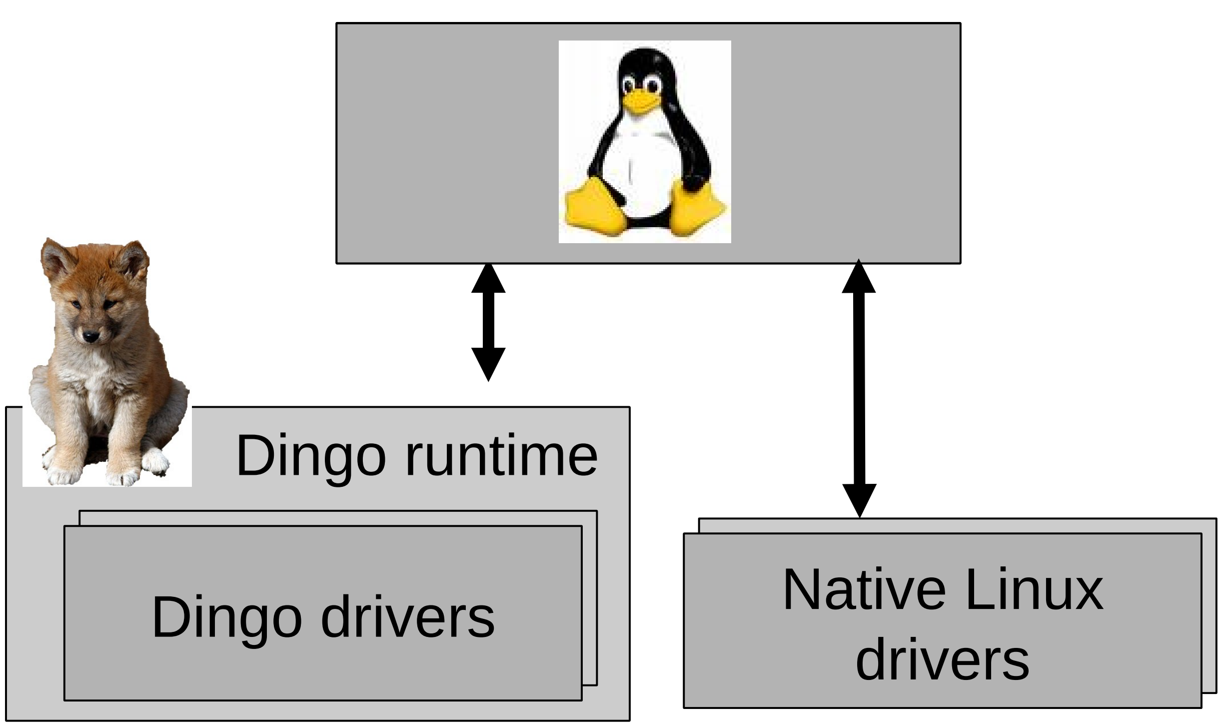
Localise complexity in driver development

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



Reduce bugs by improving the design of this interface

Dingo for Linux



A study of driver bugs

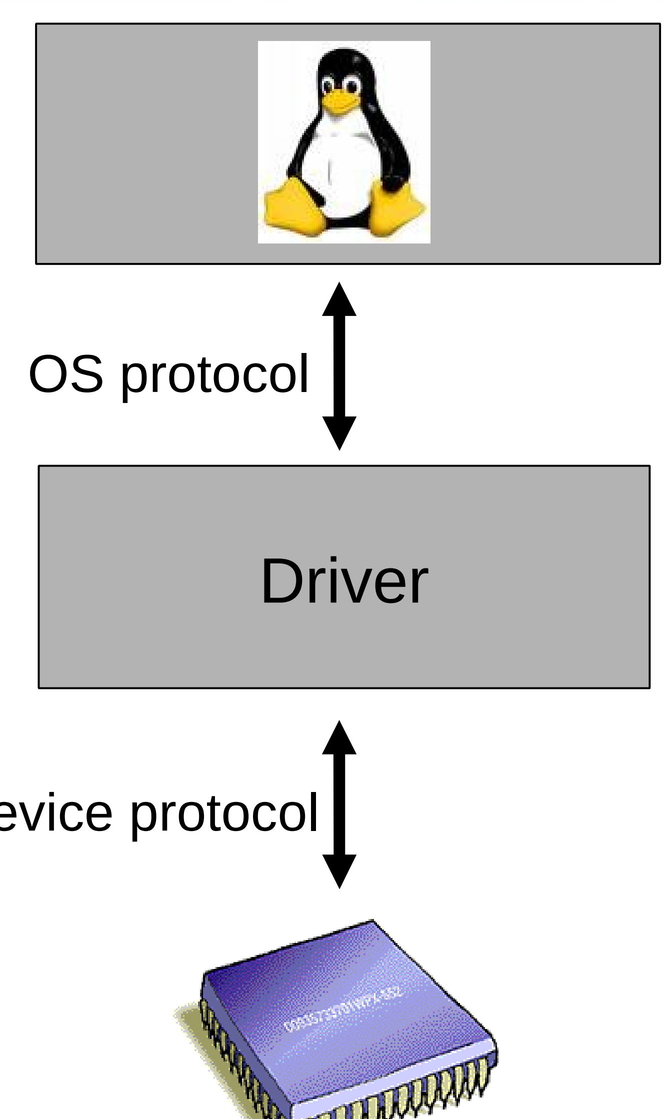
Driver defects

- Types of driver defects
  - Device protocol violations
  - OS protocol violations
  - Concurrency defects
  - Generic programming defects

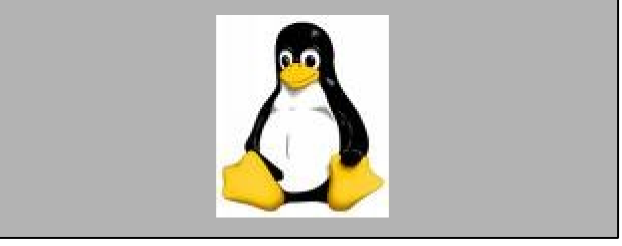
A study of Linux driver bugs

Driver	#loc	#bugs
USB		
RTL8150 USB-to-Ethernet adapter	827	16
EL1210a USB-to-Ethernet adapter	710	2
KL5kusb101 USB-to-Ethernet apapter	925	15
Generic USB network driver	1028	45
USB hub	2234	67
USB-to-serial converter	989	50
USB mass storage	803	23
Firewire		
IEEE1394 Ethernet controller	1413	22
SBP-2 transport protocol	1713	46
PCI		
Mellanox InfiniHost InfiniBand adapter	11718	123
BNX2 Ethernet adapter	5412	51
i810 frame buffer	2920	16
CMI8338 audio	2660	22
		498

A study of Linux driver bugs



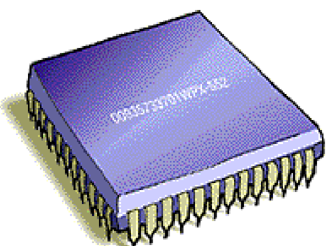
A study of Linux driver bugs



OS protocol

Driver

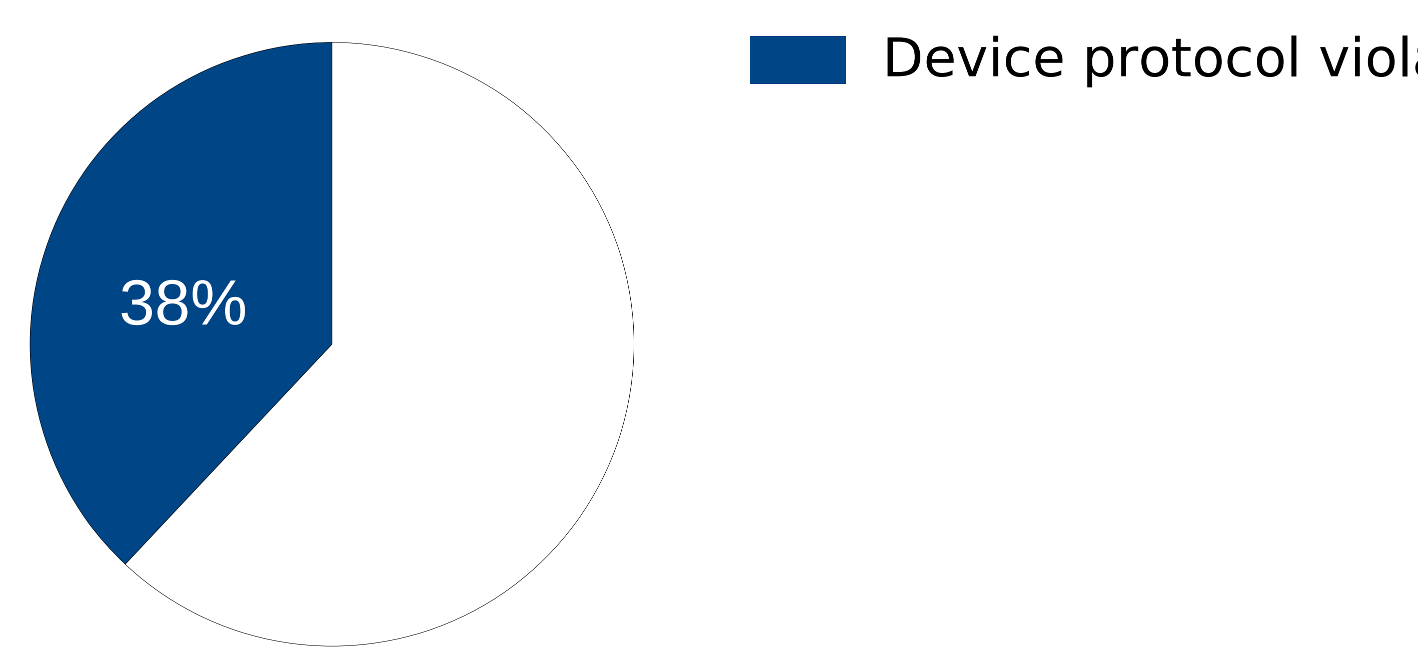
device protocol



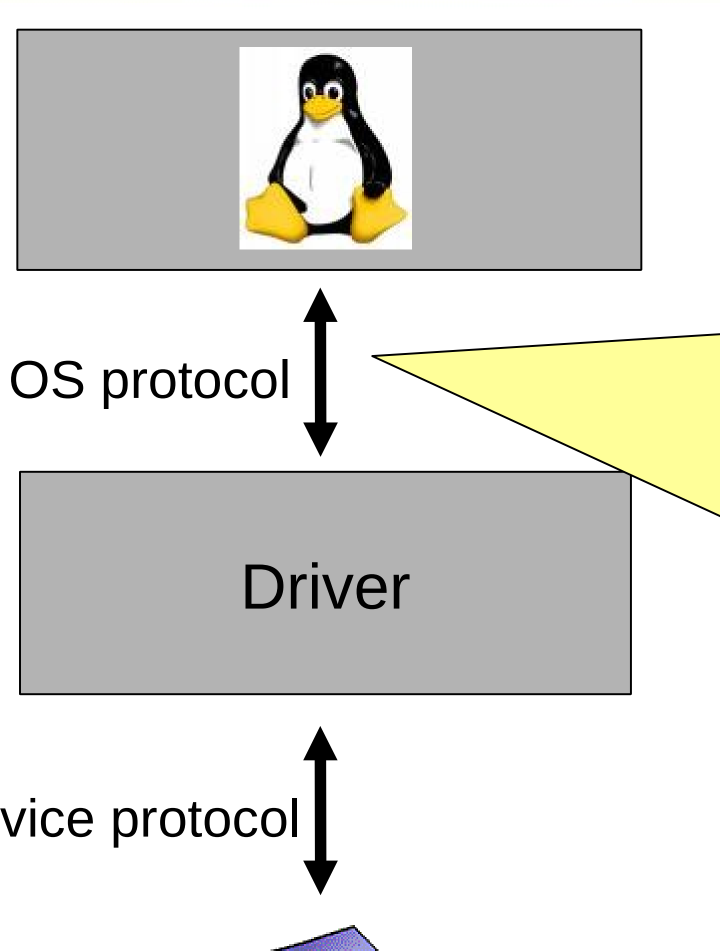
Device protocol violation examples:

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

Device protocol violations



OS protocol violations



Mellanox Infinihost controller driver

READY RESET

```
if(cur_state==IB_RESET &&
   new_state==IB_RESET){
  return 0;
}
```

OS protocol violations

38%

20%

- Device protocol violations
- OS protocol violations

Concurrency errors

Race in config functions:

Race in hot unplug handler:

Deadlock in an atomic context:

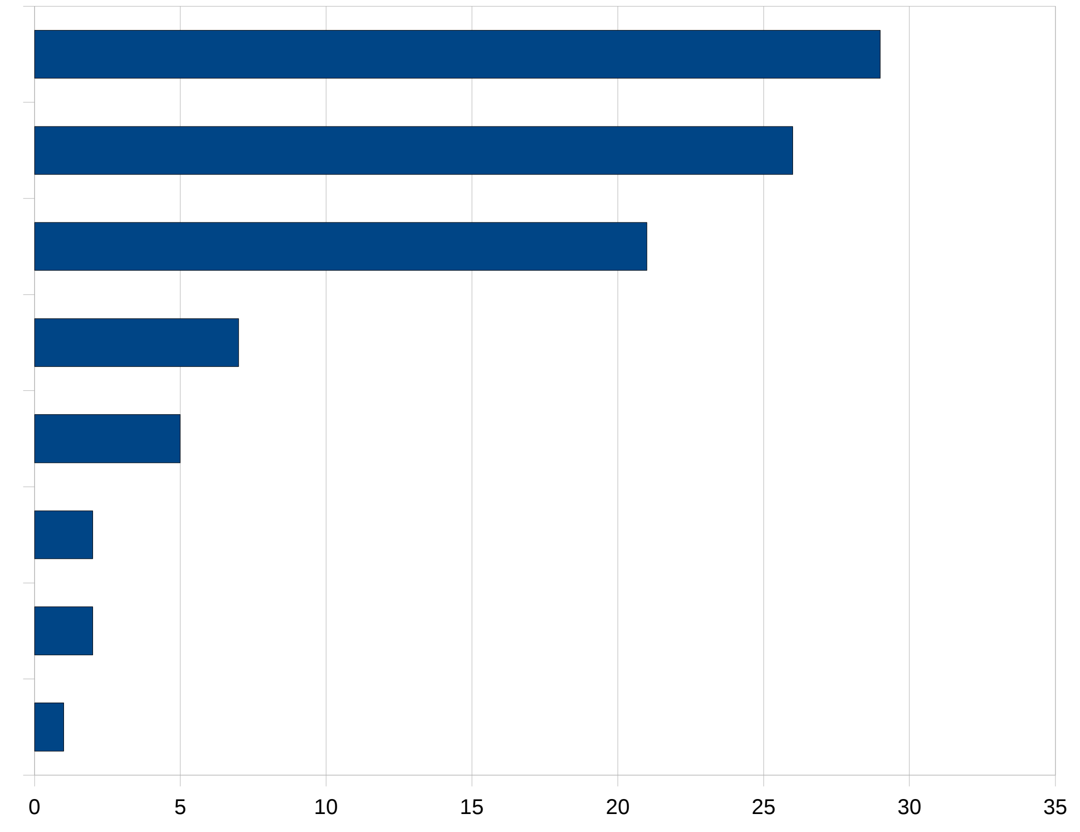
Race in the data path:

Race in PM functions:

Uninitialised lock:

Imbalanced locks:

Other:



Concurrency errors

Category	Count
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:	

0 5 10 15 20 25 30 35

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:

0 5 10 15 20 25 30 35

Concurrency errors

38%

20%

19%

Device protocol violations

OS protocol violations

Concurrency errors

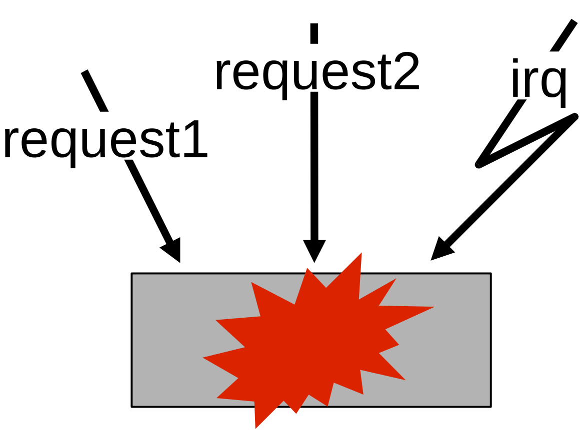
Generic errors

Category	Share
Device protocol violations	38%
OS protocol violations	20%
Concurrency errors	19%
Generic errors	23%

Dealing with concurrency bugs

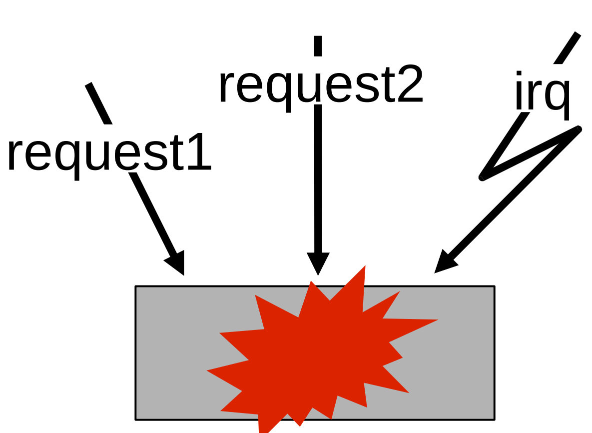
Dealing with concurrency bugs

Threads

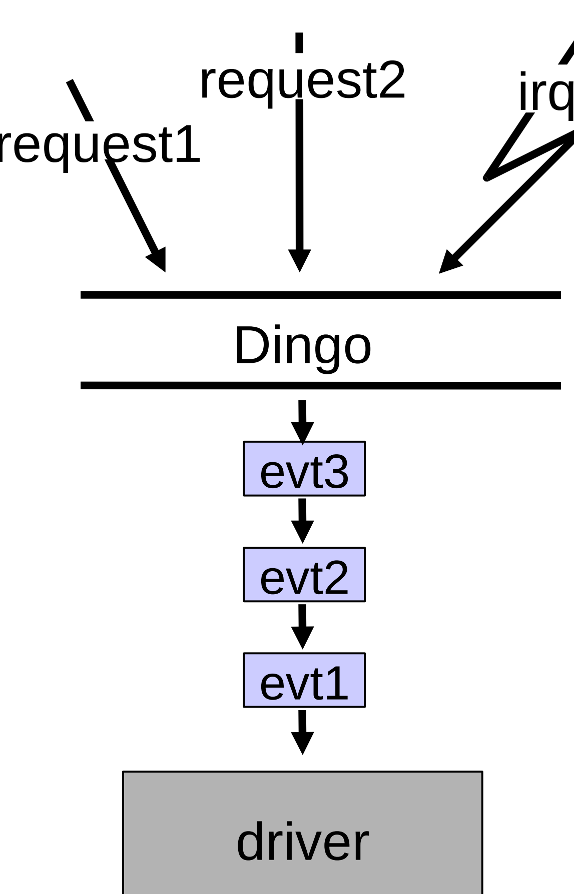


Dealing with concurrency bugs

Threads



Events



Writing non-blocking drivers

Linux

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

Dingo

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

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

Writing non-blocking drivers

Linux

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

Dingo

```
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 (%)

50
40
30
20
10
0

Linux
Dingo

1 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32

Number of Connections

Round-Trip (µsec)

800
600
400
200
0

1 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32

Impact of serialisation on performance

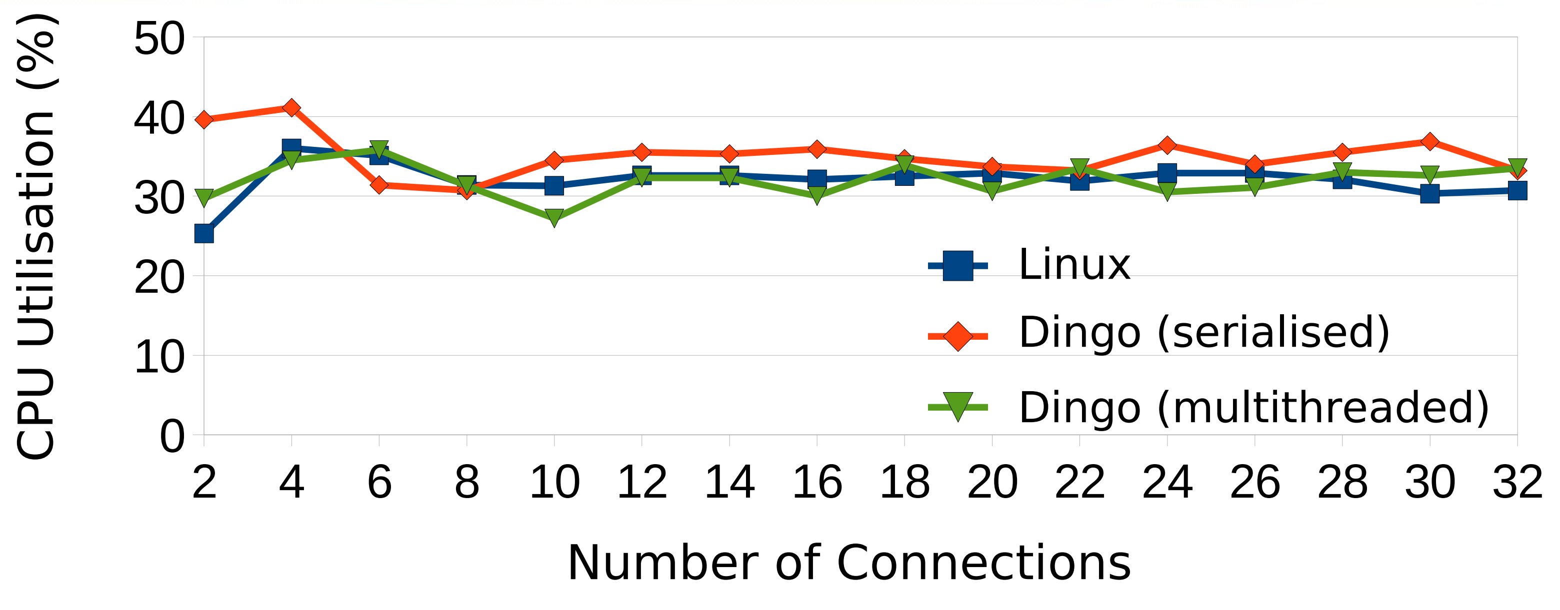
Special case: drivers for very-high-performance devices

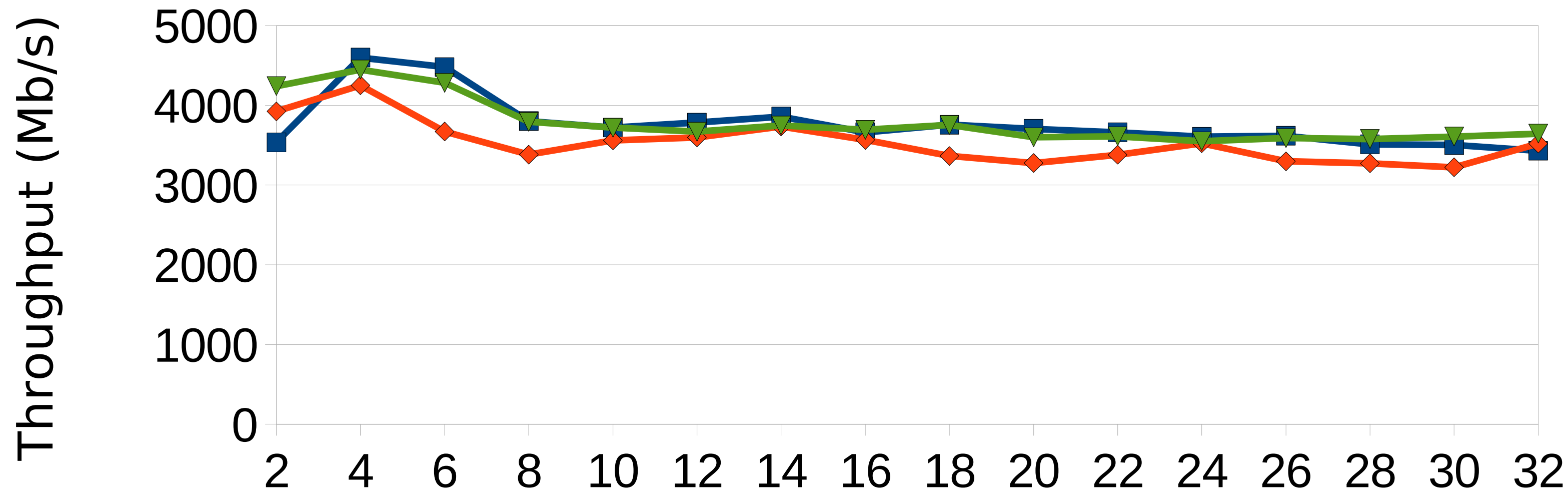
- Examples: 10Gb Ethernet, Infiniband
- For such drivers, serialisation affects performance on multiprocessors

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

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

init

?start

start

?unplugged

!startComplete

running

?unplugged

unplugged

?stop

stop

?unplugged

!stopComplete

!stopComplete

Ethernet controller protocol fragment

enabled

!enableComplete

?enable

enable

txq_stalled

!txStartQueue

!txStopQueue

txq_running

disabled

?transmit

rx

!disableComplete

disable

?disable

?receive

?suspend

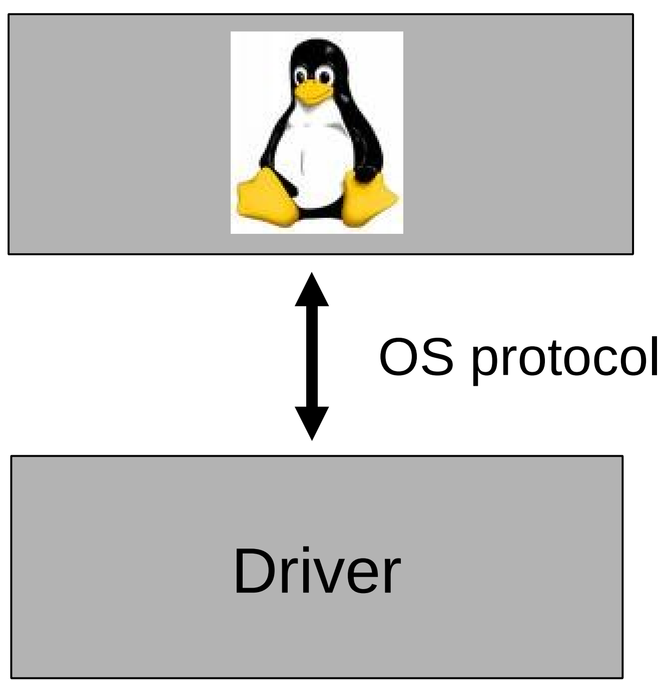
...

Other features of the language

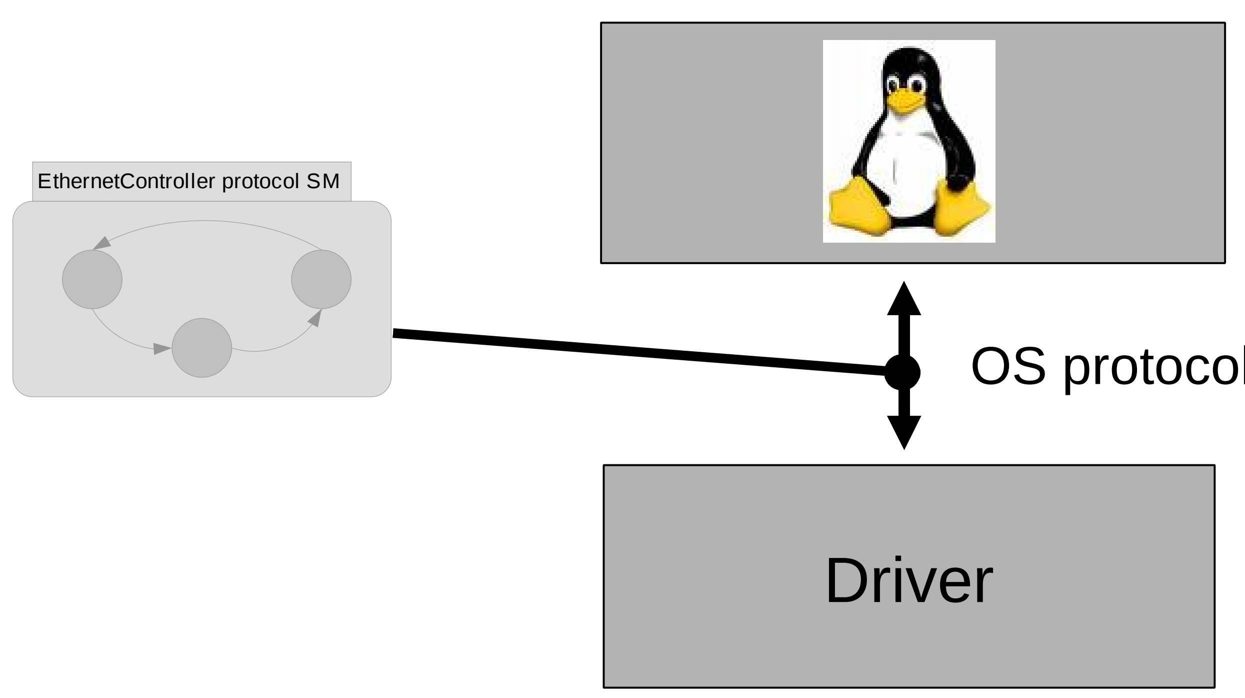
Other features of the specification language:

- Timeouts
- Protocol variables
- Dynamic protocol spawning
- etc.

Runtime failure detection



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

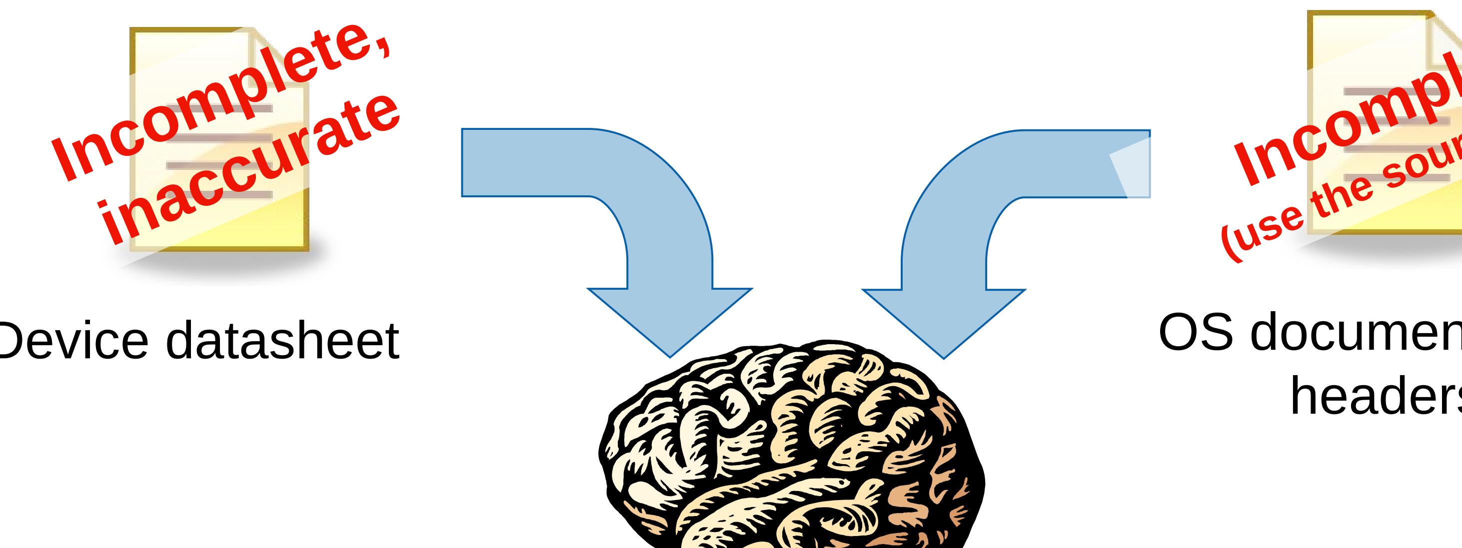
Device datasheet

OS documentation/ headers

```
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 implementation

Device drivers today




```
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 implementation

Device drivers today

Incomplete, inaccurate

Device datasheet

Incomplete (use the source, Luke)

OS documentation/ headers

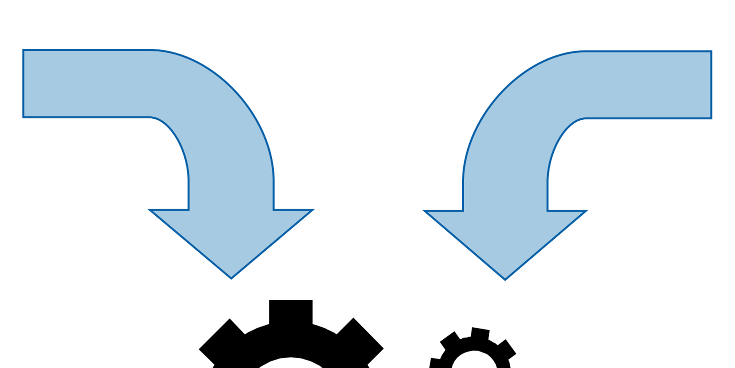
```
if (request_irq(i8042_ports[I8042_AUX_PORT_NO].irq, i8042_interrupt,
    ...
    return -1;
free_irq(i8042_ports[...].irq, i8042_... aux_cookie);
```

Drivers have more errors/LOC than any other OS component (by an OOM)

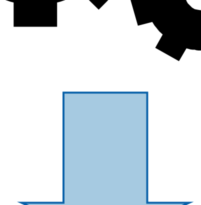
Driver implementation

Driver synthesis: a high-level view

Formal
device protocol
specification



Formal
driver/OS protocol
specification



```
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 implementation

Driver synthesis: a high-level view

Formal device protocol specification

Formal driver/OS protocol specification

```
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);
```

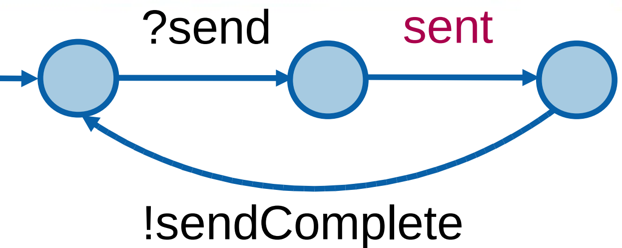
Driver implementation

Driver synthesis by example

dummy-net device

0=off 1=on	
ctrl	data

Driver synthesis by example

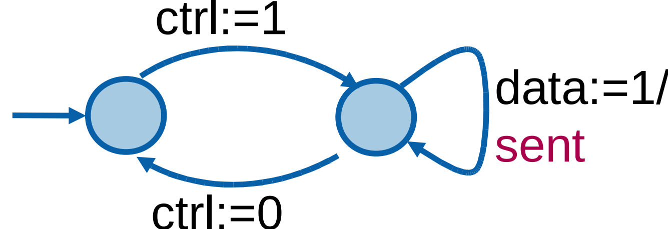


dummy-net device

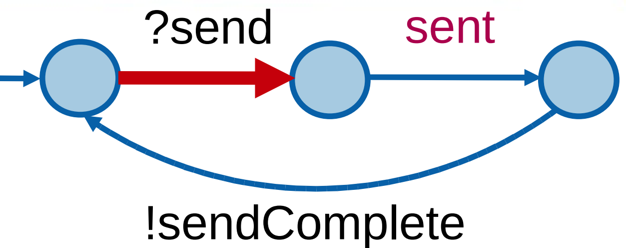
0=off 1=on	
ctrl	data

dummy-net driver

Device protocol
specification:



Driver synthesis by example

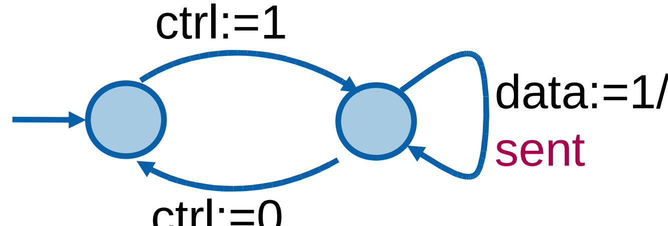


dummy-net device

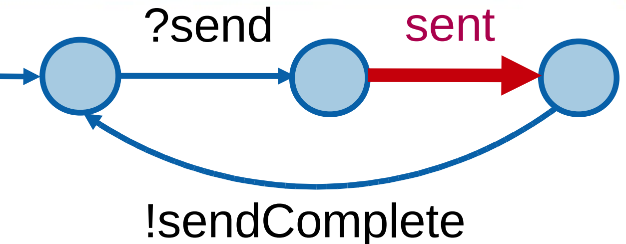
0=off 1=on	
ctrl	data

dummy-net driver

Device protocol specification:



Driver synthesis by example

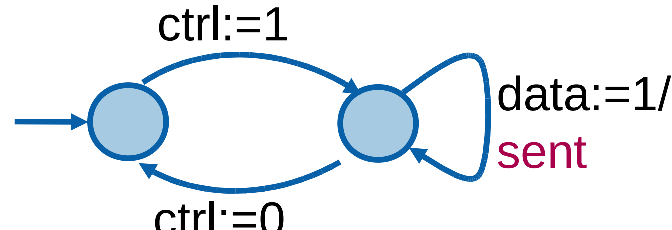


dummy-net device

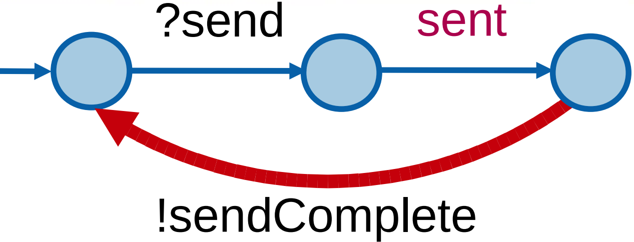
0=off 1=on	
ctrl	data

dummy-net driver

Device protocol
specification:



Driver synthesis by example

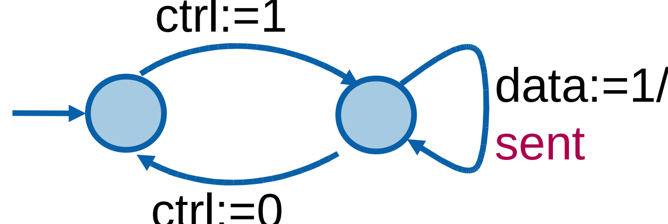


dummy-net device

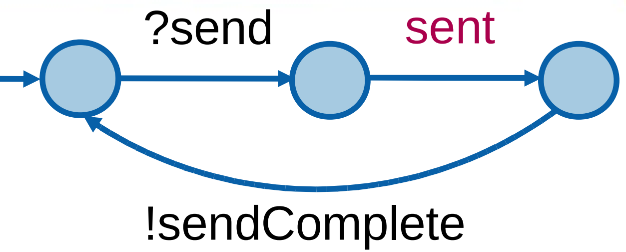
0=off 1=on	
ctrl	data

dummy-net driver

Device protocol
specification:



Driver synthesis by example

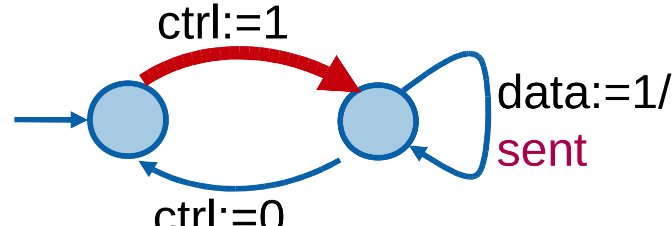


dummy-net device

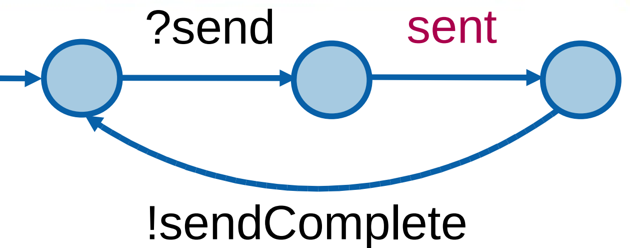
0=off 1=on	
ctrl	data

dummy-net driver

Device protocol specification:



Driver synthesis by example

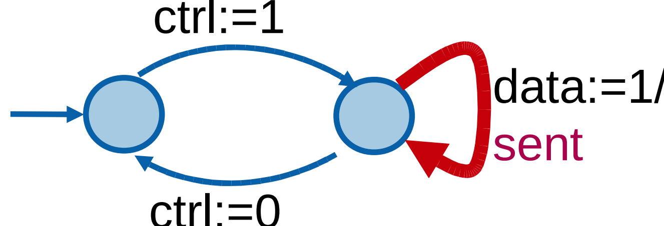


dummy-net device

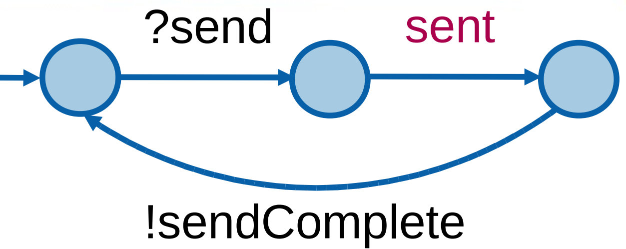
0=off 1=on	
ctrl	data

dummy-net driver

Device protocol specification:



Driver synthesis by example

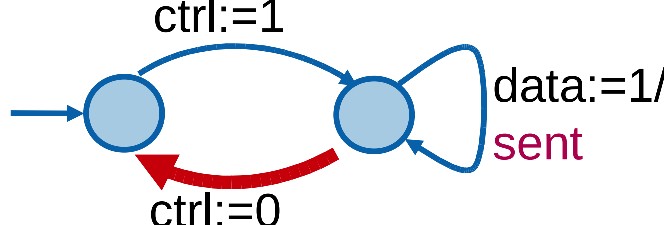


dummy-net device

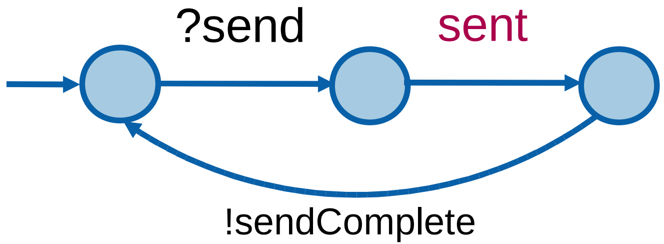
0=off 1=on	
ctrl	data

dummy-net driver

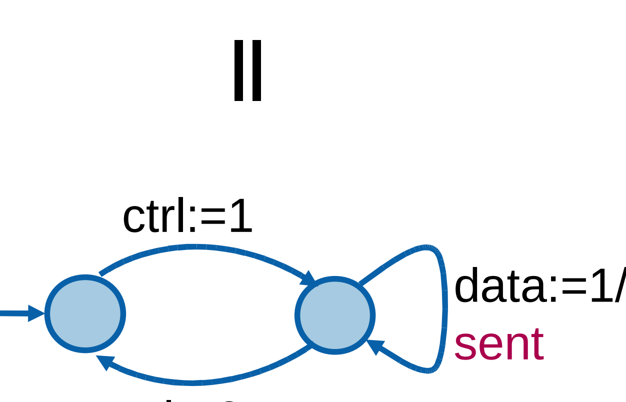
Device protocol
specification:



Driver synthesis by example



||



Driver synthesis by example

Driver synthesis by example

?send sent

!sendComplete

||

ctrl:=1

data:=1/
sent

ctrl:=0

!sendComplete

?send

ctrl:=1 ctrl:=0 ctrl:=1 ctrl:=0 ctrl:=1 ctrl:=0

data:=1/
sent

?send

!sendComplete

Driver synthesis by example

?send sent

!sendComplete

||

ctrl:=1

data:=1/
sent

ctrl:=0

?send

ctrl:=1

data:=1/
sent

?send

!sendComplete

Driver synthesis by example

?send(1) sent(1)

!sendComplete

||

ctrl:=1

data:=1/
sent(1)

ctrl:=0

?send(1)

ctrl:=1

data:=1/
sent(1)

?send(1)

!sendComplete

Driver synthesis by example

?send(1) sent(1)

!sendComplete

||

ctrl:=1

data:=1/
sent(1)

ctrl:=0

?send(1)

ctrl:=1

data:=1/
sent(1)

?send(1)

!sendComplete

Driver synthesis by example

?send(1) sent(1)

!sendComplete

||

ctrl:=1

data:=1/
sent(1)

ctrl:=0

?send(1)

ctrl:=1

data:=1/
sent(1)

?send(1)

!sendComplete

Driver synthesis by example

?send(1) sent(1)

!sendComplete

||

ctrl:=1

data:=1/
sent(1)

ctrl:=0

?send(1)

ctrl:=1

data:=1/
sent(1)

?send(1)

!sendComplete

Implementation

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

Questions?