University of Glasgow | School of Computing Science

Dependable Device Drivers

Advanced Operating Systems (M)
Lecture 12

Lecture Outline

- Sources of bugs in device drivers
- Improving device drivers: engineering approaches
  - Apple MacOS X I/O Kit
- Advancing the driver model
  - Microsoft Singularity operating system

Sources of Bugs in Device Drivers

Name	Description	Total faults	Device prot. violations	S/W protocol violations	Concurrency faults	Generic faults
USB drivers						
rtl8150	rtl8150 USB-to-Ethernet adapter	16	3	2	**7**	4
catc	el1210a USB-to-Ethernet adapter	2	**1**	0	**1**	0
kaweth	kl5kusb101 USB-to-Ethernet adapter	15	1	2	**8**	4
usb net	generic USB network driver	45	**16**	9	6	14
usb hub	USB hub	67	**27**	16	13	11
usb serial	USB-to-serial converter	50	2	17	13	**18**
usb storage	USB Mass Storage devices	23	7	5	**10**	1
IEEE 1394 drivers						
eth1394	generic ieee1394 Ethernet driver	22	**6**	**6**	4	**6**
sbp2	sbp-2 transport protocol	46	**18**	10	12	6
PCI drivers						
mthca	InfiniHost InfiniBand adapter	123	**52**	22	11	38
bnx2	bnx2 network driver	51	**35**	4	5	7
i810 fb	i810 frame buffer device	16	4	**5**	2	**5**
cmipci	cmi8338 soundcard	22	**17**	3	1	1
	Total	498	189 (38%)	101 (20%)	93 (19%)	115 (23%)

Can we address these through improvements to the supporting infrastructure for device-drivers?

Summary cause of bugs found in Linux USB, Firewire (IEEE 1394), and PCI drivers from 2002–2008
[from L. Ryzhyk *et al.*, “Dingo: Taming device drivers”, Proc. EuroSys 2009, DOI 10.1145/1519065.1519095]

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

Causes of Bugs in Device Drivers

- 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?

Type of faults	**#**
Ordering violations	
Driver configuration protocol violation	16
Data protocol violation	9
Resource ownership protocol violation	8
Power management protocol violation	8
Hot unplug protocol violation	5
Format violations	
Incorrect use of OS data structures	29
Passing an incorrect argument to an OS service	19
Returning invalid error code	7

Table 2. Types of software protocol violations.

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

Table 3. Types of concurrency faults.

[from L. Ryzhyk *et al.*, “Dingo: Taming device drivers”, Proc. EuroSys 2009, DOI 10.1145/1519065.1519095]

Improving Device Drivers: Engineering

- Some issues can be solved with good software engineering practice
  - Device drivers generally fit some hierarchy
    - E.g., a Broadcom Ethernet adaptor *is an* Ethernet adaptor *is an* IEEE 802 network interface
  - If implemented in an object-oriented language, can encode much of the common logic for a particular class of devices into a superclass which is 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
  - Can emulate in a C-based kernel, but with high syntactic and semantic overhead [see next slide – Linux does this for some driver classes]

Digression: Faking OO Code in C

```
struct vtableFile {
    void  (*delete)(void *self);
    int   (*open  )(void *self, char *filename, int mode);
    int   (*close )(void *self);
    int   (*read  )(void *self, char *buffer, int buflen);
    int   (*write )(void *self, char *buffer, int buflen);
};

typedef struct {
    struct vtableFile *vtable;
    FILE              *file;
} File;

File *newFile(void)
{
    File *f   = malloc(sizeof(File));
    f->vtable->delete = ...
    f->vtable->open   = ...
    ...
    f->file   = ...
    return f;
}
```

```
File *f = newFile();
f->vtable->open(f, "example.txt", RDONLY)
...
```

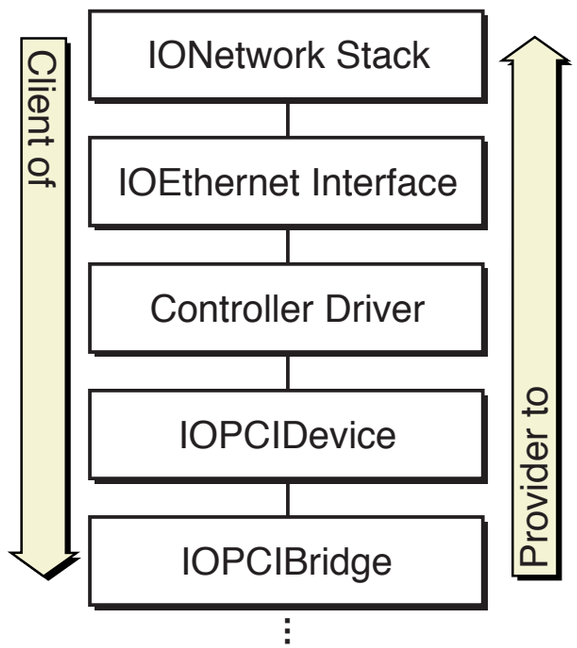
```
val f = new File()
f.open("example.txt", RDONLY)
```

Limitations of Object-Oriented Approach

- Integration with existing kernels is difficult
  - Must either emulate object-oriented approach in C, losing much of the benefits; or run dual-language kernel, with drivers in a different language to the rest of the kernel
    - MacOS X I/O Kit is an example of the latter
- Abstracting logic into a common framework doesn’t address bugs in that framework

Example: MacOS X I/O Kit

- Object-oriented framework for building device drivers in MacOS X
  - Devices organised into *families*, represented by C++ abstract classes
  - Drivers for particular hardware device types are concrete classes, implementing the abstract classes for their family
  - Drivers are instantiated as *nubs* that represent individual devices
  - Implemented using a restricted subset of C++ – without exceptions, templates, multiple inheritance, or RTTI since these are too complex to implement efficiently and safely within the kernel



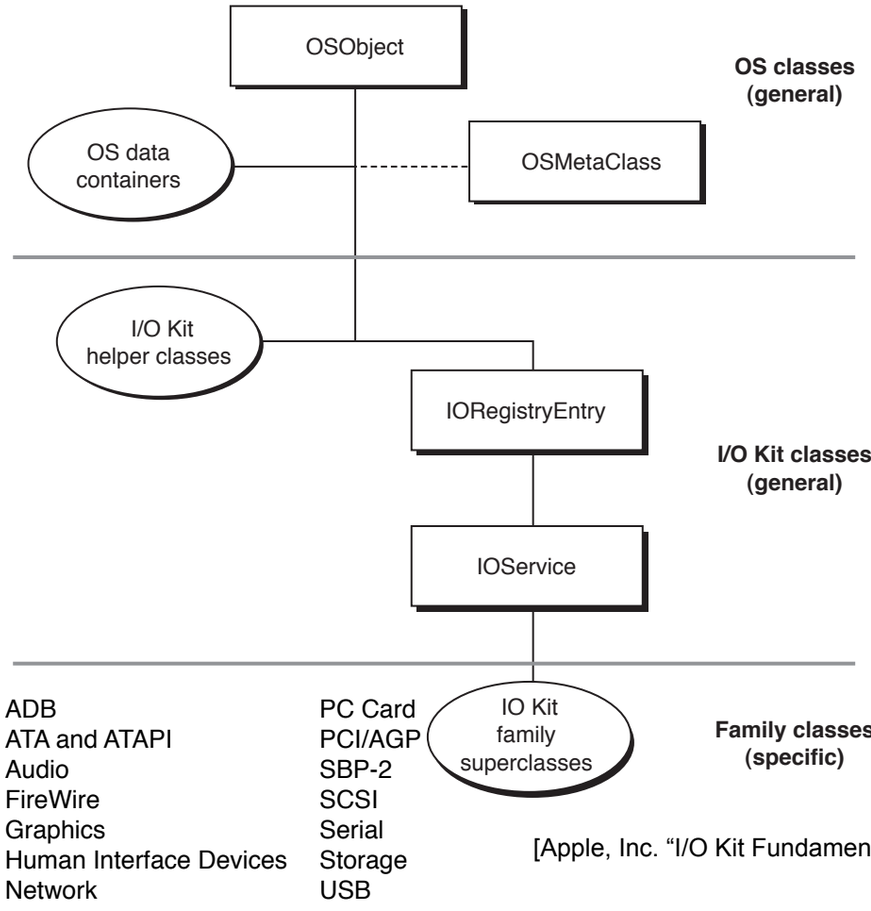
[Apple, Inc. “I/O Kit Fundamentals”, 2007 http://developer.apple.com/library/mac/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/IOKitFundamentals.pdf]

- Layered driver model

- Provides lifecycle management for devices and their resources
- Simplifies event handling and work loops, abstracting out a common model for devices and device families

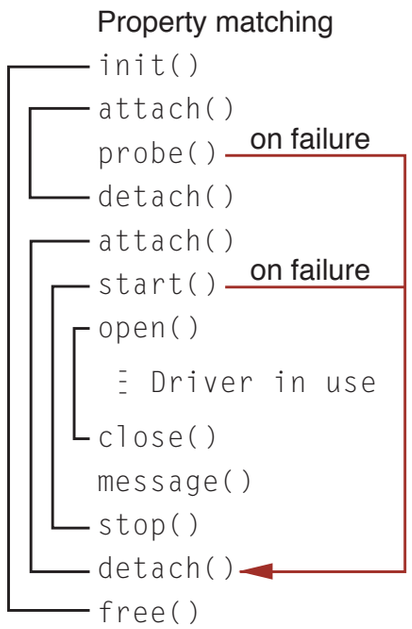
I/O Kit – Objects and Families

- Many common functions and boilerplate are abstracted into family-specific classes
- Families provide standard facilities to help a device manage data during I/O operations – DMA scatter-gather lists, virtual address translation, etc.
- Provides a “robust system for protecting access to driver resources during I/O operations, which frees driver writers from having to write their own code to enable and disable interrupts and manage locks on the driver’s private resources”.



[Apple, Inc. “I/O Kit Fundamentals”, 2007]

I/O Kit – Lifecycle Management



[Apple, Inc. “I/O Kit Fundamentals”, 2007]

- All drivers follow a common basic lifecycle, defined by the methods of the `IOService` class
  - The figure on the left shows the lifecycle methods
  - Families and individual device drivers override these methods as needed; each must call the corresponding method in its superclasses; progressive refinement
  - A driver with no need for special behaviour can just accept the inherited functionality, and not define these methods
- Other methods provide for power management, messages for device status changes, etc.

- Coding the lifecycle in a common superclass of all device drivers in this manner ensures consistency
  - The compiler forces that drivers that don’t implement a particular method to inherit a sane default – a kernel where the object hierarchy is manually implemented in C requires the programmer to explicitly manage the vtable entry, leaving scope for bugs

I/O Kit – Events and Work Loops

- Device drivers can be accessed by multiple concurrent threads
  - Multiple user processes
  - Interrupts from the hardware
  - Timeouts, power management events, and other system activity
  - Asynchronous callbacks from device or user code
- Locking can be hard to manage
- Yet, the underlying hardware is generally single threaded – e.g., you can only send one Ethernet packet at once
- Rather than each driver manage its own concurrency, the kernel translates system calls and other actions into *events*, posted to a per-driver *work loop*

- Drivers in this model are logically single threaded
  - Each work loop has one or more objects of type `IOEventSource`
    - Use `workLoop->addEventSource()` to add an event source to the work loop
    - Various subclasses of `IOEventSource` exist for different event types (e.g., interrupt, timer)
  - Callback functions are registered with these event sources, and automatically called with appropriate locks held when events occur; concurrency is managed by the kernel
    - The parameter to the callback indicates which event has occurred
- Moves complexity into the kernel, but greatly simplifies driver code

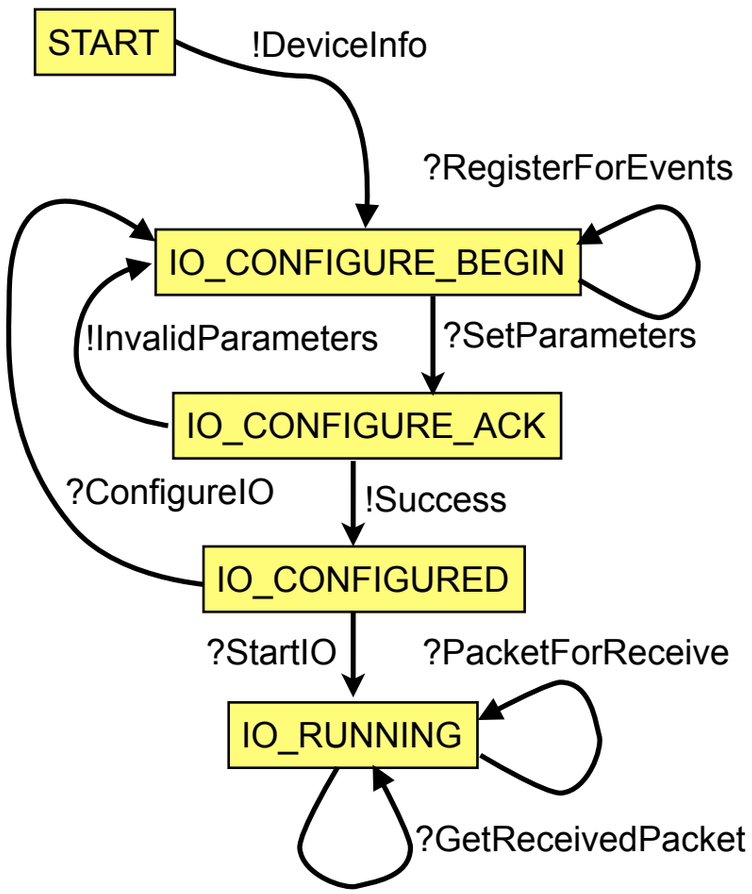
Improving the Device Driver Model

- Interaction between device driver and OS can be represented as a finite state machine

- Can formally model these state machines
  - Explicit is better than implicit
  - Incorporate formal descriptions of the states, transitions, and events into the code and type system
  - Enabled correctness of the state machines to be checked

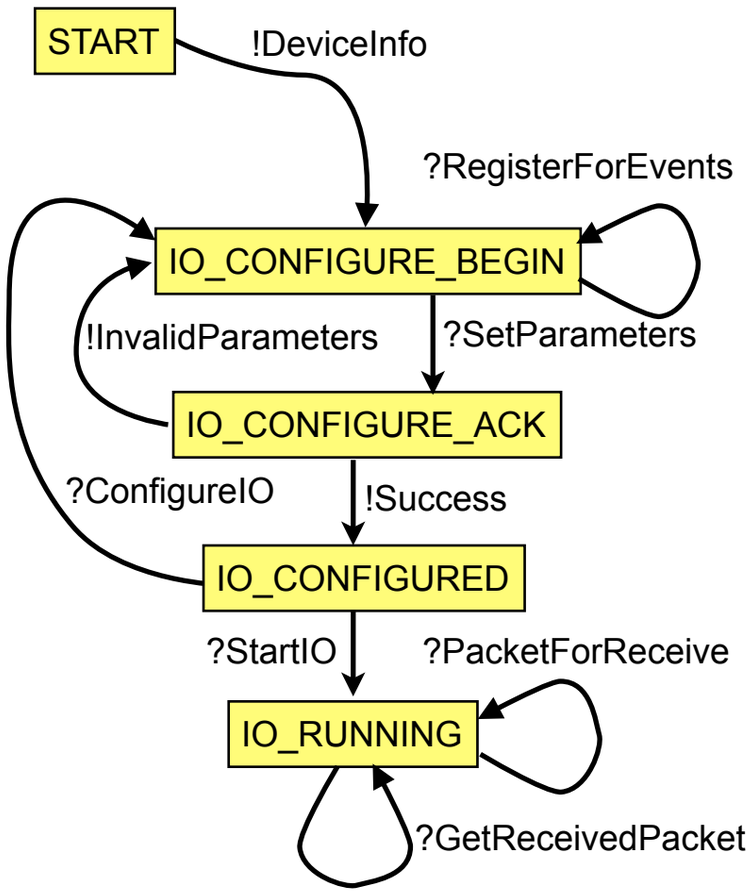
- Document the assumptions and requirements in a format that can be verified automatically

Modelling State Machines



- A set of *states* and *transitions* triggered by/causing *events* form a state machine
- The MacOS X I/O Kit models incoming events to a driver, but not the states, allowable transitions, or generated events
- We could formally define the full state machine in the source code
  - List of states
  - List of events that can be received in each state
  - For each event that can be received, what is the next state, and what events are generated in response
- Could be implemented by annotating methods in a Java-like language – or by extending the language
  - Compiler or stand-alone verification tool can then check that the code implements the defined state machine

Example: Singularity – State Machines



- System comprises a set of concurrent processes that communicate solely by exchanging messages
- A *contract* defines the state machine for a process
- Implemented in Sing# – an extension to C# – the compiler can check that the contract is implemented by processes declaring their support

```
contract NicDevice {
  out message DeviceInfo(...);
  in  message RegisterForEvents(NicEvents.Exp:READY
  c);
  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.

[G. Hunt and J. Larus. Singularity: Rethinking the software stack. ACM SIGOPS OS Review, 41(2), Apr. 2007. DOI 10.1145/1243418.1243424]

Example: Singularity – Pattern Matching

- Contract defines the state machine – essentially an abstract type
- Implementation uses pattern matching against received messages
  - A function for each state
  - Each function switches based on the type of the message object received

```
NicDevice.Exp:IO_RUNNING nicClient ...

switch receive {
  case nicClient.PacketForReceive(buf):
    // add buf to the available buffers, reply
    ...

  case nicClient.GetReceivedPacket():
    // send back a buffer with packet data if available
    ...

  case nicClient.ChannelClosed():
    // client closed channel
    ...
}
```

the state

messages that can be received in that state

[M. Fähndrich *et al.* Language support for fast and reliable message-based communication in Singularity OS. Proc. EuroSys 2006. DOI 10.1145/1218063.1217953]

- Compiler checks that `switch receive` statements handle all messages defined by the contract
  - Blocks in the switch receive statement must end with a transfer of control, to a function representing a new state or to itself, allowing compiler to check transitions

- Messages are immutable objects
  - Simplifies locking – no need to lock the message, just the message passing code

Verification of State Machines

- If the state machine is formally defined in the code, we can begin to verify it
  - Check that the code implements the defined state machine
  - Check the state machine itself
    - Validate that the driver cannot deadlock
    - Validate that certain states can be reached
    - …
    - [discussed further in the MRS4 course]

- Contracts in Sing# can readily be translated into (fragments of) a Promela model, suitable for verification with a model checker such as SPIN

Event-driven *vs.* Concurrent Models

- Two models for driver state machine code
  - Concurrent model – Singularity
  - Event-driven model – MacOS X I/O Kit, Dingo [see reading at end]
- Different ways of expressing the same concept
  - Apple had valid engineering reasons to prefer an object-oriented event-driven model – familiarity, and ease of integration
  - The concurrent model used in Singularity conceptually cleaner, but requires kernel structured for light-weight concurrency and message passing [will return to this in later lectures]

Summary

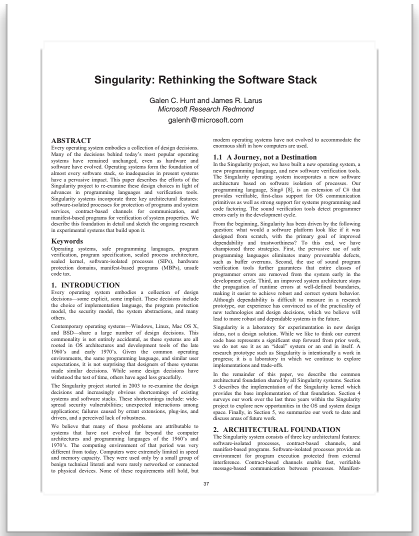
- Most operating systems employ an ad-hoc device driver model
  - Significant numbers of driver bugs are due to poor specification and documentation of this model
  - Good software engineering practices can improve this somewhat, while integrating with existing kernels
- A clean-slate design can explicitly make the state machine underlying the driver visible
  - Allows automatic verification that the driver implements the state machine for its device class
  - Allows model checking of the state machine

Further Reading

- L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. Proceedings of the European Conference on Computer Systems, Nuremberg, Germany, April 2009. ACM/EuroSys. DOI 10.1145/1519065.1519095

Dingo: Taming Device Drivers

- G. Hunt and J. Larus. Singularity: Rethinking the software stack. ACM SIGOPS OS Review, 41(2), Apr. 2007. DOI 10.1145/1243418.1243424



Singularity: Rethinking the Software Stack

Galen C. Hunt and James R. Larus

Microsoft Research Redmond