Hotplug in a Multikernel Operating System

Submitted by: Animesh Trivedi

August 19, 2009

Under supervison of
Adrian Schüpbach, Dr. Andrew Baumann
and Prof. Timothy Roscoe

Systems Research Group
Department of Computer Science (D-INF)
Swiss Federal Institute of Technology, Zürich (ETHZ)

Abstract

The future manycore architectures present serious challenges to operating system designers. The traditional operating system designs can no longer manage the capabilities and power of the diverse heterogeneous cores with complex memory hierarchies, interconnects resembling networks, and distributed I/O configurations. The Barrelfish multikernel operating system addresses these issues by treating hardware as a distributed system. Cooperation in such an environment is achieved by message passing. We borrow similar design ideas in this thesis and present a *distributed* USB hot plugging infrastructure for a multikernel Barrelfish operating system. We have divided the USB system into three primary modules with different responsibilities: Host controller driver, USB manager and client drivers. This modular design provides necessary isolation and flexibility required in many-core systems. It also provides freedom to schedule and, if required, migrate any module independently among the cores depending upon the system workload and the application requirements. These modules communicate by explicit message passing but a few frequently updated and performance critical data structures are shared using shared memory mechanism. In this thesis, we try to design, implement and evaluate this system on top of message services and abstractions provided by the Barrelfish operating system.

Acknowledgments

I would like to thank my mentor Prof. Timothy Roscoe for giving me the opportunity to work in Systems research group at ETH, Zürich. Over the past two years his constant motivation and enthusiasm for research has always inspired me to do better.

I would like to thank Dr. Andrew Baumann for his guidance and help while developing the work presented in the thesis. I have learned a lot from him during the course of our interaction in past few years.

I would also like to thank Adrian Schüpbach for his constant support, motivation, and inputs that I had during this thesis. His prompt response to my problems and long debugging emails had definitely helped me getting out of some tricky situations.

In the end, I would like to thank people of D-INF at ETHZ who directly or indirectly have helped me to get at this point.

Contents

List of Figures

List of Tables

Chapter 1

Introduction & Problem Statement

Hardware landscape has changed tremendously over the past decade. Intel is envisioning 'Tera-scale' computing [1] and AMD is making heterogeneous chips by putting GPU and CPU in a same package [2]. Things have moved from a single fast processor to the heterogeneous manycore architectures; from a flat uniform access memory to the non-uniform memory access; from a centralized device management to the distributed device management. With hardware getting more diverse, a system starts to resemble like a network and factors such as quality of service in system interconnect, congestion, hop counts, delays, hot plugging (akin to node joining or failure), rapid prototyping and testing are getting more important. This hardware diversity has three major aspects [3]:

- **Non-uniformity:** Apart from the much discussed non uniform memory access (NUMA) architectures, multiple level of cache sharing is also common in manycore processors. Hence a flat uniform shared memory model may not be the best way to model this complex memory hierarchy. Additionally the actual cost of accessing memory depends depends upon interconnect topologies like Hypertransport and Quickpath, and factors such as hop count and routing play an important role.

- **Core diversity:** In near future the processing core will be much more diverse and heterogeneous. The system will have more number of specialized cores with different performance, power trade offs and possibly different instruction sets.

- **System diversity:** With the internal system components getting diverse, it will be difficult to write and optimize system software about any specific target. With a network like interconnect topology the access to devices in the system also depends on which core driver code is running. Even today, AMD systems have two PCI root complex and cores near root complex have faster access to the devices than other cores sitting on the other side of the bus.

Barrelfish [4] is a new operating system, designed specifically for handling future manycore heterogeneous architectures. It is a multikernel operating system, which treats computer hardware as a distributed system where a node represents an entity inside the computer box such as CPU, devices, GPU etc. Nodes communicate much like as done in a distributed environment by passing messages on previously agreed protocols.

Handling dynamic plugging and unplugging of devices in such a distributed environment imposes some interesting challenges such as where to run code for the drivers, how to manage workloads, how to reflect status of the devices to the rest of system etc. Modularity in a big system like USB is an important issue. With three different kinds of controllers and plenty of devices around it is desired to have a clean design which is flexible and can provide necessary isolation and protection. With system internals getting much more diverse it is also important to think about abstraction provided to the higher layers.

In this thesis we present USB hot-plugging infrastructure for Barrelfish operating system. The thesis is organized as follow. Chapter 2 gives an overview of related device driver research work. Chapter 3 introduces and talks about Barrelfish operating system and its interaction with device drivers. Chapter 4 and 5 give an overview about USB protocol and Intel EHCI controller hardware. In chapter 6 we introduce our distributed USB infrastructure. Chapter 7, 8, and 9 talk in detail about USB memory management, host controller driver and USB manager respectively. In chapter 10 we will provide a brief overview of the USB mass storage protocol and details about our implementation of the device driver for flash mass storage device. Chapter 11 gives detail about evaluation and performance. In the chapter 12 we present some conclusions and we end in chapter 13 after giving some directions about the future work.

Chapter 2

Device Drivers

Device driver is a piece of software which allows operating system kernel and other higher level programs to interact with a hardware device. Device drivers make up majority of operating system's code base, which requires support for myriad of devices. Also, these are the most buggy and unreliable part of an operating system code. In Windows XP, drivers cause 85% of reported failures [5]. Padioleau et al. [6] gives a comprehensive overview of driver evolution in the Linux kernel.

2.1 Improving Reliability of Device Drivers

Device drivers have drawn considerable amount of attention from systems research community to improve reliability of commodity operating systems. These efforts can be break down into following sections depending upon their level of application:

2.2 By Better Design Choices

In a traditional operating system model, device drivers are run into kernel space. Hence a faulty driver can corrupt internal kernel data structures and can bring whole system down. To cope up with such situations kernel designers have advocated moving drivers out into the user space from kernel space and to facilitate communication through inter process communication (IPC) primitives [7, 8]. Often due to an inefficient implementation, this design exhibits poor performance [9]. Though [10] has shown to achieve good performance with the user space driver designs. This approach can not be applied to already deployed legacy driver code base which would require a complete rewrite.

Driver virtualization helps in isolating the buggy device drivers from rest of the systems and hence improves its stability. In driver virtualization device drivers (often unmodified or with little engineering efforts) are run in separate isolated virtual machines [11, 12]. It provides better driver re-usability with strong isolation. At the same time it invokes trust issues between the kernel and virtual machines. Also drivers operating in different virtual machines raise timing and lock synchronization discrepancies. Addendum, poor performance in such systems (specially in server workloads) can [13] be a limiting factor in their deployment. Twindriver [14] approach tries to address this issue by semi-automatically deriving *hypervisor drivers* from guest OS drivers. It runs performance critical operations directly into the hypervisor and coordinates with the guest OS instance of the driver. It uses Software Virtual Memory (SVM) mechanism to let hypervisor instance efficiently access the driver data in guest OS address space while protecting hypervisor address space from corrupted memory access from guest OS driver.

Chipounov et al. [15] tries to reverse engineer a driver for better reliability and reasoning, by monitoring the interaction between a device and the driver. It combines these traces with static analysis of the driver to derive a protocol state machine. This machine is then used to synthesize new drivers from pre-verified safe driver templates. New drivers are safe by virtue of construction because only *safe* set of traced data was used to extract the protocol.

Microdriver approach [16] offers a trade off between an in-kernel driver's performance and an user level driver's isolation. It factors a driver into two parts: a performance critical part and another non-critical part. Performance critical code is then moved to the kernel space and non-critical code is run in user space. I/O data transfer logic, which often resides in performance critical code paths, is moved into kernel space. Non-critical path usually contains device management and configuration, error handling etc. Both instances communicate and synchronize to complete a request. User mode driver helps in driver isolation and error containment. Also driver developers can get a helpful support from the traditional developing, testing and debugging tools and libraries, which often are absent when developing in-kernel drivers. Though a faulty in-kernel driver part can still corrupt kernel data structures. To automatically split driver code into kernel and user space driver, Microdriver requires manual annotations from developers to properly identify the shared data structures.

2.3 By Improving Reliability and Driver Restart

To address the issue of in-kernel data corruption, Nooks [5] provides a in-kernel subsystem that helps in preventing such errors. Nooks executes in-kernel extensions such as device drivers and loadable file-system modules etc. in an isolated *lightweighted kernel protection* domain. The domain still is in kernel mode but with a restricted write access to the critical in-kernel data structures. A new reliability layer keeps tracks of all the modifications made by an extension and traps any unwanted changes, such as invalid arguments for a kernel service or consumption of too many resources in the system. Upon error detection or failure, recovery logic is triggered. Nooks was able to detect and isolate 99% of errors injected in a testing Linux system. Though Nooks is not able to differentiate between a faulty and a malicious driver. A driver (malicious) still can corrupt system state by executing privileged instructions. It provides very basic kind of recovery mechanism by securely freeing all allocated resources to the failed driver and then restarting it. It relies on virtual memory hardware support for isolation within kernel. Also, it does not support any user level tools during development of a driver which could have been helpful in detecting basic control flow errors such as infinite loops etc.

Much of the effort in the direction of reliable operating systems has been put to make OS kernel immune to a faulty in-kernel driver. In such systems if a driver crashes, then kernel survives but not the driver applications. From an application's perspective either both, kernel and driver, should work or none. Having a surviving lone kernel is not much of a use to an application, which lost data in driver failure. Shadow driver [17] provides an elegant way to transparently restart the driver. It inserts a shadow driver *tap* between kernel and the driver, and passively monitors all the communication between these two. Upon detection of a failure, it actively replaces the failed driver with a shadow driver and triggers the restarting mechanism. Shadow driver is not a true driver as it does not provide complete driver services but it is used to hide driver failure from applications by impersonating the communication between applications and the driver. Once failed driver has been restarted shadow driver delegates the pending requests and control over to it. Shadow driver uses kernel support for dynamic loading and unloading of drivers as its primary mechanism to restart a failed driver. It assumes failed driver subsystem has *no side effects.* This assumption may be not true for stacked drivers. A failed EHCI controller or a PCI driver can cause failure to many other driver systems in a cascading effect. It also may have limited applicability because it requires explicit communication among driver, applications and kernel for tapping. It is not possible to deploy it for the class of drivers which use shared memory mechanism for communication such as video drivers etc. Also due to timing discrepancies it is ill-suited for applications with real time demands.

XFI [18] suggests a software guard for protection of system address space but suffers from poor performance and lacks control flow safety checks.

2.4 By Analysis and Verification

Many researchers argue to use a type safe language for driver or kernel extension development. But because of the high overhead and less control these solutions are less appealing to the driver development communities. SafeDrive [19] provides a way to do type checking on existing systems which are coded in C. It uses annotations in C to define semantics and does fine grained memory safety checking. Pointers and no-annotation in legacy code can be problematic for its deployment. To overcome these limitations [20] proposes the use of a *Secure Virtual Architecture* (SVA), which performs run time monitoring and checks based on LLVM.

[21] uses the same technique as of Microkernel architecture by moving device drivers into the user space but it goes one step further by validating their references. It uses device safety specification (DSS), which describes state and transition of driver state machine. All the references from the driver including memory, I/O ports, interrupts, registers etc. are validated through reference validation mechanism (RVM). If a driver is caught attempting to execute an illegal operation, it then is stopped and restarted. Though approach sounds promising but defining and clear state machine for complex devices may require considerable amount of efforts. Also doing *unsafe* driver restart is not desirable in high availability systems.

Due to frantic pace at which modern devices are being developed it is hard to catch up with them. To improve reliability of hardware operating code for devices Devil [22] attempts to check and verify device register manipulations before actually writing them to devices. Devil is an interface definition language (IDL) which upon giving device communication interface and related semantic specifications, generates safe hardware operating code to operate the device. Devil lacks, run time assembly checks, bus architecture and communication protocol considerations (like PCIe and USB, packet passing in USB), new features such as power management etc. Ryzhyk et al. [23] proposes to formalize the communication interface between device driver and an operating system kernel. Their solution is based on a formal state machine based verification, which helps in enforcing correct driver behavior. It requires a major rewrite of device drivers and extracting state machine information from device specification could be an error prone task. Similarly, Dolev et al. [24] have also proposed a notion and requirements for a self stabilizing device drivers based on a state machine approach.

Ball et al. [25] proposes to perform a comprehensive static analysis of the device drivers. They emphasize on proper kernel API usage. Their static driver verifier (SDV) tool uses a static analysis engine (SLAM) with *API usage rules* to find kernel API abuses. It reduces the program control flow into a boolean program abstraction which preserves control flow of the C code. Then system performs symbolic model checking to ensure that the program obeys the API usage rules. It is quite useful to find many programming glitches before actual deployment of the driver. On the other hand it completely ignores the driver behavior and implementation related bugs and focuses only on API usage aspect of it. Due to close integration with operating system (in this case, Windows platform) and its API documentation, it would require significant efforts to port it to other systems. DDVerify [26] attempts to port technique introduced by it to Linux with further enhancements such as verifying concurrent software with shared memory.

As a pure language based solution, Spear et al. [27] proposed a design to verify and reason about the correctness of a device driver. Drivers are written in a type safe language. To prevent corruption to system state, drivers are run in user space. Hardware resources are only accessed through messages, and when granted access from the verification system. Verification system relies on meta-data resource declarations for carrying out assessment of a request. It relies on the type safe language capabilities to restrict malicious behavior of a driver code.

Chapter 3

Barrelfish

The system hardware is changing at a much faster pace than before. The execution cores are getting heterogeneous in terms of power requirements, instruction sets, speed, and capabilities. The flat uniform memory assumption on which traditional operating system designs are built is no longer valid. Barrelfish aims to manage and exploit power and capability of such future manycores system.

Barrelfish [4] is a multikernel operating system. The kernel design is inspired by the microkernel design architecture [7, 9]. The Barrelfish kernel only runs a set of minimal services which are required to run in privileged mode such as scheduler, message sending services etc. All other responsibilities are managed by special user space service servers like memory manager, file system manager etc. By lowering the foot print of the kernel one can also argue about correctness and reliability of the system. The system design is built around three design principle. First, make all intercore communication explicit. It removes OS design reliance on the shared memory model, which is a bottleneck point in the system. It also improves overall understanding of the system as now operating system knows *what*, *when* and *who* about system state updates.

Second, have minimum OS design dependency on hardware. The only architecture specific code in Barrelfish is message transport mechanism and interface to hardware (CPU and device drivers). Hence the higher layer protocols and algorithms are isolated from the low level hardware details.

Third and the last, replicate instead of share. Many system states like dispatcher queues are shared among processors to get a consistent view of the system. These shared data structures act as contention points and restrict system scalability. To counter this issue Barrelfish replicate system state between core instead of sharing. Consistency is maintained by message passing.

3.1 Barrelfish and device drivers

Like in any other operating system (OS) device drivers are responsible for handling devices in Barrelfish. This new 'distributed' environment presents many interesting challenges for driver developers as well as to the OS in terms of efficient and optimized resource usage. As shown and discussed in [3], in a system with network like interconnect (AMD NUMA machines) the cost of accessing a device and memory depends upon on which core the driver is running. The difference is significant in terms of memory access. Hence it is *desirable* to do a topology aware resource allocation for the drivers for better resource usage and performance. Drivers which run on cores which have direct access to device and associated data buffers in memory are expected to perform well in such systems. As shown by [4], with changes in spatial configuration of network card, driver and memory buffers, the UDP goodput was decreased from 887.9 Mbps to 502.7 Mbps.
To manage complexity of modern systems, Barrelfish makes use of a *system knowledge base (SKB)*. It contains representation and information about machine's hardware and current state. It can be queried by other services in system to get knowledge about:

1. Resource discovery, such as a new USB device is attached.
2. Online measurement and profiling, for things like available bandwidth, latency or power on USB. It may be probed to check if with current configuration USB system is able to satisfy requirements of a newly connected device.
3. A priory knowledge about a device or controller, from data sheets [28].

Such queries *can* answer about the appearance and disappearance of devices, their capabilities and requirements etc. Hence pushing conventional driver architecture towards declarative systems. Declarative systems are more reliable and one can reason about their correctness unlikely as in imperative systems. Also it helps in topology aware device management logic. For example, a USB device can put it requirements in terms of polling frequency, delay threshold, bandwidth etc and then controller after consulting SKB about current tree topology can assert whether it will be able to full fill these requirements or not. Similarly, it could provide suggestions about device relocation (like in systems with multiple USB ports), whenever possible, for better performance.

In Barrelfish, device drivers are run as user level processes in their own separate execution domain. Hence a buggy driver can not crash down whole operating system. Device access is done by memory mappings. OS services are provided by the *service servers* like PCIe server and clients communicate with them by message passing. In such a distributed environment, USB driver *should* be developed as a distributed driver with different functionality such as configuration, device management, data transfer etc. running on different cores. To make a better match with current system state and driver requirements one can factor a driver in performance critical and non-critical segments similarly to the Microdriver architecture [16]. Performance critical piece of code can be deployed on cores near to the device. Driver also can specify their requirements (much like in [27]) prior to their initialization so that an appropriate core can be selected. I/O buffers (and their management logic) which often are in performance critical paths can be allocated in RAM of cores which are nearer to the device. Other things like configuration management, error recovery etc. which are executed occasionally can be run on other under utilized cores. A periodic evaluation system, which upon a requirement or request can relocate these pieces to different cores, if approved by underlying decision making architecture.

3.2 Service Servers & Flounder

Like in a distributed environment the services in Barrelfish are provided by service servers. Anything from frame allocation, memory mapping, interrupt delivery, device discovery etc. are provided by specific servers. The requests are send and response is received by explicit message passing. Example of few service servers are memory server, PCI server, EHCI controller server, SKB server etc.

Barrelfish maintains a name-server called *Chips* which is a centralized name server registry. A new server has to register its services and name to the Chips. All message passing are done on names of servers. Server clients can search specific server name using SKB, if appropriate entries are made into it.

Since message passing mechanism is vital and used intensively in Barrelfish, a domain specific language (DSL) *Flounder* is developed for hiding these explicit message passing mechanisms. Flounder takes the service interface specifications and automatically generates associated server and clients stubs. These stubs help in communicating with Chips for service registration, message passing between server and clients, registering service handlers, and invoking the handler when an associated message is received. More details can be found in Flounder manual [29] at Systems Research Group, ETH.

Chapter 4

USB System Overview

This chapter gives a brief overview of Universal Serial Bus (USB) 2.0 protocol. It also talks about key elements in USB architecture and how data is moved across the USB bus. For complete reference please refer to the official USB documentation [30]. The chapter highlights the related sections in the documentation for cross reference purposes.

4.1 Hot Plugging

Hot plugging or Hot swapping is a technology which enables user to replace, remove or add system components to a running system without shutting it down. Once the appropriate software/driver is installed on the system, user can plug and play the attached peripheral without rebooting the system. Hot plugging is a desirable property in the systems with high availability. It enables the system maintainer to remove faulty component or add additional peripherals for redundancy without affecting the running system. Hot plugging examples include, a faulty RAID disk can be replaced by a new one with using hot plugging, USB mass storage devices, USB wireless cards etc.

4.2 USB System

The universal serial bus (USB) is a complex bus design to connect a host computer to a number of different peripheral devices. It aims to replace various wide range and slow buses like parallel, serial etc. with a standard bus which can be used to connect variety of devices such as keyboards, mouse, printer, web cam, storage devices etc. The latest revision of USB is 2.0 (High speed) which supports bandwidth up to 480 Mbps (or 60 MBps) and 127 devices per host controller. The next revision, 3.0 is under development and expected to support bandwidth in the range of 5 Gbps [31].

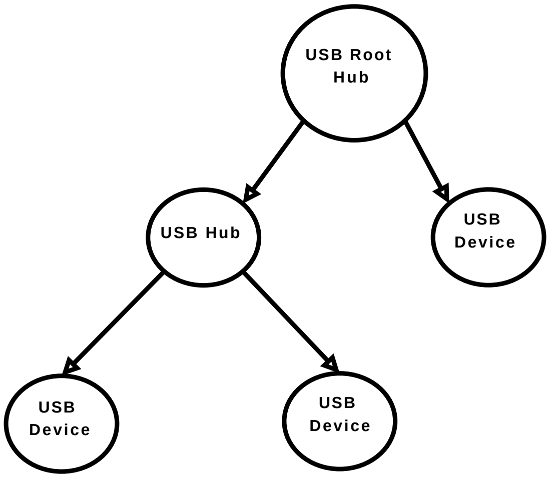

Figure 4.1: USB connection topology

The design goals of USB system include:

- A single connector to connect any PC peripheral
- A method of avoiding system resource conflicts
- Hot plug support
- Automatic detection and configuration of the USB devices
- Low cost
- Enhanced performance capabilities
- Support for legacy devices
- Low power consumption

A typical USB system consists of a host controller, hubs and devices. USB bus topology is laid out as a tree. Host controller acts as the root and others, hubs and devices, act as internal and leaf nodes respectively, which are connected through several point to point links. Hubs can be seen as expansion slots which helps in expanding the USB tree as shown in figure 4.1. The hub uses a master-slave protocol for communicating with the devices. Hence every kind of communication (in or out) is initiated by the host controller. By choosing the master-slave protocol USB does not have to do a distributed bus arbitration to avoid collision on the shared resource.

4.3 USB Hardware

The USB hardware present the part which is actually implemented in the silicon. A complete USB system has three major hardware components:

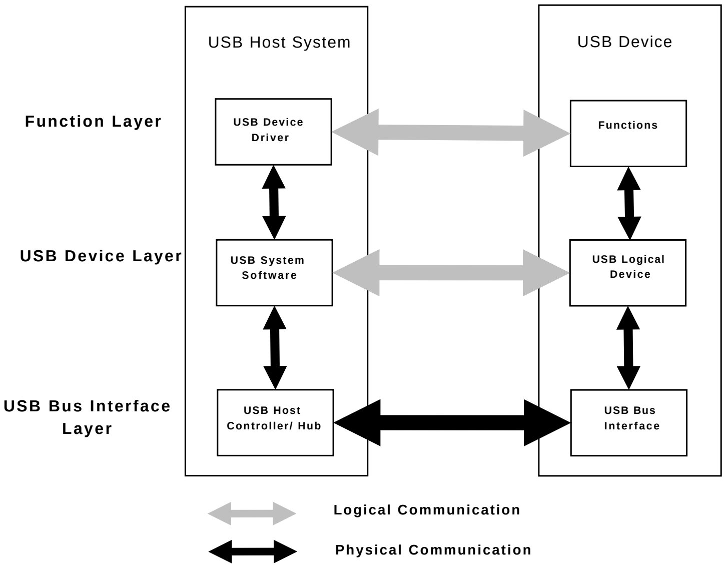

Figure 4.2: Communication flow in a USB system

4.3.1 USB Host Controller

The host system hardware contains a USB host controller which is responsible for initiating the transaction over the USB system under host driver control. The three generations of controllers have been around which are:

1. Universal host controller interface (UHCI) (USB 1.0)
2. Open host controller interface (OHCI) (USB 1.1)
3. Enhanced host controller interface (EHCI) (USB 2.0)

The host controller reads the list of USB transactions and executes them. If a write to a device has been requested then controller will read data from buffer supplied from USB client driver and deliver it to device. In case of read, when device sends data back to host controller, it forwards it to buffer supplied by USB client driver.

4.3.2 USB Hubs

USB hubs are like attachment points for the USB system where user can insert a device or even another hub. They are responsible for enabling/disabling the ports, maintain status of ports and notifying the host controller in case of events such as attachment/removal of device happens on the port. Every USB host contains a root hub which is a central attachment point and all USB traffic originates from it.

4.3.3 USB Devices

USB devices are special devices which can understand USB protocol. Any device functionality can be ported to USB protocol given bandwidth, delay requirements etc. are met. They are powered and configured after plugging into the system by the USB driver. Every USB device has a *device descriptor* which is presented to host controller at the time of configuration. It contains information regarding features and capabilities required to operate the device and help finding corresponding client driver for it. An USB device can be a low (1.x), full (1.x) or high (2.0) - speed device. An USB device has only one device descriptor but can have multiple *configuration descriptors* corresponding to device's power or bandwidth requirements. Going further, a configuration descriptor can have multiple *interface descriptors*, which themselves can have multiple *endpoint descriptors*. A single USB device can present itself as more than one device having different interfaces e.g. a web cam with a microphone or speakers, connected through a single USB connection to host system. Such USB device will require different client drivers depending upon which interfaces have been enabled on the device.

An endpoint acts like a data source or sink. The endpoint descriptor provides information related to endpoint's transfer type, direction (IN or OUT) and data rate supported. A pipe is a logical entity that connects endpoint to the client driver. Pipe can be of two type:

- **Message pipe:** Content which is delivered through message pipe is required to have standard USB defined structure. Transfer on message pipe is done in a three step process, consists of a request, optional data and status phases. Message pipes allow communication in both direction. Default control pipe, which is used to configure devices, is always a message pipe.

- **Stream pipe:** Stream pipes deliver data with no USB required structure on data content. Stream pipes are always unidirectional in content flow. Stream pipes support bulk, isochronous and interrupt transfer types.

Section 5.3.2 in USB documentation discusses in detail about pipes and their types.

4.4 USB Software

The USB software system consists of following three layers:

4.4.1 USB Client Driver

USB client or device driver is the highest layer in the software hierarchy. It is responsible for implementing a typical device functionality such as keyboard, mouse, web cam, USB mass storage etc. and often is unaware of underlying complexity of USB architecture. It issues I/O requests to USB bus driver in terms of I/O Request Packets (IRPs).

4.4.2 USB Bus Driver

The USB bus driver understands the USB bus topology and target device characteristics. It also keeps track of device bandwidth and power requirements. When an IRP is received from USB client driver, it breaks the request into USB transactions which will be executed on the next frame. A frame is a bus time quanta on which different transactions are scheduled. A typical USB 1.x frame is 1 ms and corresponding USB 2.0 frame (often called *microframe*) is 125 μsec.

A USB bus driver is responsible for:

- Device configuration management
- Data transfer management
- Bus management

4.4.3 Host Controller Driver

The host controller driver (HCD) is primarily responsible for scheduling the USB transactions over the bus. HCD maintains lists of various pending transactions (control, bulk, interrupt or isochronous) [1] and schedule them accordingly. The specifications of scheduler is implementation dependent.
According to Intel EHCI documentation [32], system software maintains two schedule for the host controller: a periodic schedule and an asynchronous schedule. The isochronous and interrupt USB transactions fall into the periodic schedule, while bulk and control transfers belong to the asynchronous list. In each microframe if periodic schedule is enabled then host controller will execute from the periodic list. It will only execute from the asynchronous schedule after it reaches the end of periodic schedule. Chapter 5 discuss in detail about EHCI hardware.

4.5 Communication over USB

An USB device client initiate a transfer when it puts I/O requests in terms of IRPs to USB bus driver. The client driver supplies a buffer required to

[1] explained in next section

hold or provide data for the transfer. The bus driver breaks the IRPs into small USB transactions which adhere to the bus and protocol requirements. A transfer happens between a client driver and an endpoint on the USB device. USB supports following types of transactions over an endpoint:

4.5.1 Control Transfer

It is used to send and request short data packets to configure an USB device. It involved reading and setting device, configuration, interface and endpoint descriptors.

A control transfer is consists of a *setup* bus transaction moving standard request information from the host to a device, zero or more *data* transactions sending data in the direction indicated by the setup transaction and a *status* transaction returning status information from a device to the host. A setup transaction is only considered complete if status returned shows ”success”.

Control transfer is done on messages pipes by sending standard USB commands codes. The data exchanged on message pipe also has USB defined structure. The section 5.5 in USB documentation discuss in detail about control transfers.

4.5.2 Bulk Transfer

It is used to send and request relatively large amount of data packets using maximum allowable bus bandwidth. Modern USB storage devices use this transfer type. Requesting a pipe with a bulk transfer type provides the requester with:

- Access to USB on a bandwidth-available basis.
- Reliable delivery.
- Guaranteed delivery but no guarantee for bandwidth or latency.

USB imposes no data content restriction on transferred data. Bulk transfer employs stream pipes hence are uni-directional. The section 5.8 gives more detail about bulk transfers on USB.

4.5.3 Interrupt Transfer

Interrupt transfer is used with those devices which need to send or receive data infrequently but with a bounded service periods. Host controller will automatically repeat these type of requests in the specified time intervals. It provides:

- Guaranteed maximum service period for the pipe.

- Retry of transfer attempt on next period in case of a failure.

It uses stream pipes and has unidirectional traffic. Section 5.7 in USB 2.0 documentation gives more comprehensive overview about it.

4.5.4 Isochronous Transfer

It is used to send or receive real time data streams with guaranteed bus bandwidth but without any reliability. These are suitable for A/V devices and signals. The traffic through these pipes is constant-rate and error tolerant. It provides:

- Guaranteed access to USB bandwidth with bounded latency.
- Guaranteed constant rate data though pipe as long as data is provided to the pipe.
- No retries on occasional failures.

Isochronous transfer uses stream pipes and is unidirectional. Section 5.6 in USB 2.0 documentation gives more details about it.

4.6 Interaction with Devices

The USB devices operate on two separate layers. The layer one can be seen as *control layer* which is responsible for maintaining the device status, processing control requests, setting up configuration and interface. The second layer is *functionality layer*. This is the actual functional capability that the device provides like a mouse, or flash storage etc.

The communication to control layer is done though message pipes and with set of standard commands which are explained in section 9.3 and 9.4 of USB 2.0 documentation. The response for these commands are also standard descriptors (device, configuration, interface or endpoint). These are explained in section 9.5 of USB 2.0 documentation.

Chapter 9 of USB documentation gives comprehensive details of USB devices framework and associated descriptors.

Chapter 5

EHCI Overview

Enhanced host controller interface (EHCI) is an USB host controller (HC) hardware specification for USB 2.0 protocol. In this chapter we will discuss about the Intel EHCI specification [32]. While discussing we will focus on essential elements of EHCI description which are necessary for better understanding of host controller driver implementation. For detailed descriptions please refer to [32].

5.1 Registers and Mackerel Interfacing

The EHCI controller has two different functional register sets. The first set called *capability registers* is responsible for laying out the capabilities of the host controller. The capability registers specify the limits, restrictions, and capabilities of the host controller implementation. These values are used as parameters to the host controller driver. It contains information such as such as number of ports, number of companion controllers (CC), routing logic, HC supports 32 or 64 bit etc. It also contains the starting address of memory region for the second set of registers called *operational registers.* The operational registers are used by system software to control and monitor the operational state of the host controller.

Mackerel [28] is device interfacing language. Mackerel compiler upon giving device interface related specifications generates C code to safely manipulate device state by reading or writing registers. Mackerel generated code is used for interacting with the HC registers. The capability register set is a variable length memory region and operational register set starts after it. Hence it is required to break single EHCI device into two separate devices: EHCI capability device and EHCI operational device. The Mackerel dev files `ehci_cap.dev` and `ehci_op.dev` contain registers specification for above mentioned two devices respectively. Upon initialization first EHCI capability device is initialized and operation register offset is read from it to initialize the EHCI operational device.

Table 5.1: EHCI capability registers

Offset	**Size**	**Mnemonic**	**Description**
00h	1	CAPLENGTH	Capability length register contains the beginning of the operation register space.
02h	2	HCIVERSION	This register contains a BCD encoding of the EHCI revision number supported by the host.
04h	4	HCSPARAMS	This register contains set of fields that are structural parameters, like number of ports etc.
08h	4	HCCPARAMS	This register contains multiple mode control control and addressing capability etc.
0Ch	8	HCSP-PORTROUTE	This is an optional register. If implemented then contains route mappings for companion controllers.

Table 5.2: EHCI operational registers

Offset	**Size**	**Mnemonic**	**Description**
00h	4	USBCMD	Contains commands to be execute by the controller.
04h	4	USBSTS	Reflects the pending interrupts and various states of the controller.
08h	4	USBINTR	Enables or disables reporting of the corresponding interrupt to the software.
0Ch	4	FRINDEX	Contains the index into the periodic frame list.
10h	4	CTRLDSEGMENT	Contains the higher [32-63] bits of EHCI data structures.
14h	4	PERIODICLISTBASE	Contains the base address of periodic frame list.
18h	4	ASYNCLISTADDR	Contains the address of next asynchronous queue head to be executed.
40h	4	CONFIGFLAG	Contains flags for port routing logic.
44h	4	PORTSC	Contains status of each USB port.

Table 5.1 and 5.2 gives and overview of the registers and their associated functions. Chapter 2 in EHCI documentation discuss in detail about these registers.

5.2 Associated Data Structures

In this section we will discuss EHCI data structures used to communicate with control, status and data between controller driver (software) and controller hardware. We will focus on how asynchronous queue management is done in EHCI. All interrupt, control and bulk data streams are managed via queue heads (QHs) and queue element transfer descriptor (qTDs). These are queued in asynchronous queue for execution.

5.2.1 Queue Element Transfer Descriptor (qTDs)

The qTDs are only used with the queue heads. These are used for one or more USB transactions. The structure block contains two link pointers for queue advancements and five data element array for buffer pointers. This structure must be physically contiguous and 32 bytes aligned. The table 5.1 shows the block diagram for the queue element layout. Please refer to the section 3.5 in EHCI documentation for detailed description.

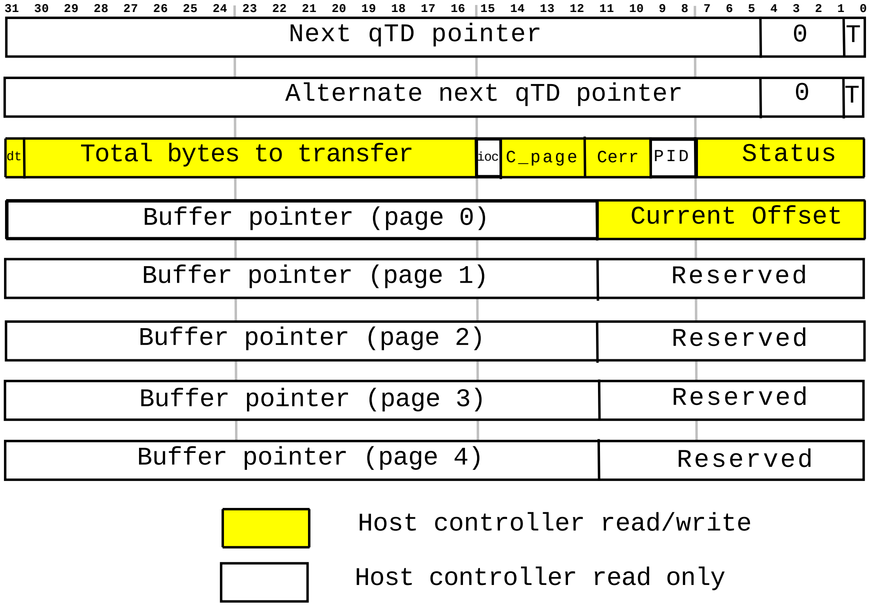

Figure 5.1: qTD data block layout

The status field reflects the current status of the queue elements. Host controller writes back the qTDs only after the transfer retirement.

5.2.2 Queue Heads (QHs)

A queue head is used to perform transfer on a particular endpoint on the USB device. A data transfer request is made to an endpoint by using a queue head and associated queue elements. A queue head can only have data transfer in one direction except on default endpoint which is bi-directional. The queue heads are linked to each other via physical address pointers to make a circular link list. This list is called asynchronous link list and controller hardware execute it in a strict round robin manner.

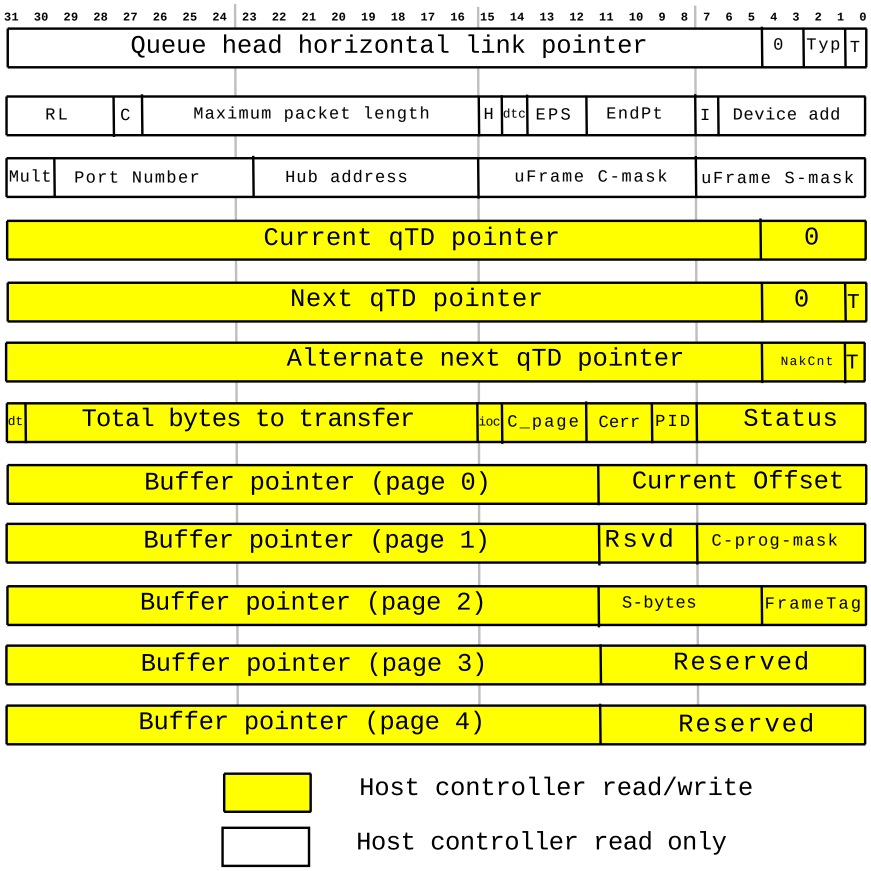

Figure 5.2: Queue head data block layout

The dword 1 in queue head contains endpoint characteristics including its address, maximum packet size and endpoint speed etc. The dword 2 contains endpoint capabilities but is mostly used in split transaction for low and full speed devices. The multiplier is used for high speed isochronous endpoints.

The host controller writable area as shown in figure 5.2 is called *transfer overlay*. Controller driver is only required to initialize the controller's readable area, except when driver is maintaining data toggle bit[1] in queue head which requires setting up 1 bit in the overlay. The overlay is automatically initialized by the queue element which hardware is executing currently. The current queue element pointer contains the address of that queue element. When the execution of queue element is finished the updated status if written back to the queue element to reflect the changes.

[1] Explained in 5.3.5

The meaning and semantics associated with every field are explained in great detail in section 3.5 and 3.6 of EHCI documentation.

5.3 Operational Model

In the previous section we explained about the data structures used to interact with EHCI hardware. In this section we will discuss how this interaction is done and how EHCI handles various responsibilities.

5.3.1 Port Routing and Control

A EHCI controller can consists of one EHCI programming interface and 0 to N USB 1.1 companion host controllers. These are required to handle low and full speed devices. When such devices are detected the responsibility of handling them is delegated to one of the companion controller. The EHCI has port and status register for every port but companion controller has only the port control and status registers which it is required to operate. The port activity notification is first sent to the companion controller, if it exists.

The routing logic can be defined in two ways: a global policy or per port manner. The global policy is set by setting *configured flag* (CF) bit in CONFIGFLAG register. Upon setting it, all port related activity notifications are sent to EHCI. On the other hand the port also can be delegated to companion controller by setting *port owner* bit in port status register. For current implementation the controller driver sets CF bit to 1 so that all notifications on port activity are routed to EHCI. Section 4.2 in EHCI documentation talks in detail how delegation and routing is done.

5.3.2 Periodic Queue

The periodic queue contains elements from isochronous and interrupt transfers. It is scheduled in every micro-frame. Hence periodic schedule provides bandwidth and latency guarantees to the application. When a new micro-frame is started controller hardware always start execution by checking periodic frame list. If there are any more pending execution then they are executed first before moving on to the asynchronous schedule. A new request is only accepted when it is feasible to schedule it in periodic list with the current workload.

The current implementation does not support this. The section 5.6 in USB 2.0 documentation and section 4.6 and 4.7 in EHCI documentation discuss it in detail.

5.3.3 Asynchronous Queue Management

Asynchronous queue is where all the bulk and control transfer is managed. The host controller executes this list only when it reaches the end of the periodic schedule. The asynchronous list is a simple circular list of queue heads. The register AYNCLISTADDR register contains the pointer to the next queue head. Hence all en-queued queue heads are executed in a strict round robin manner.

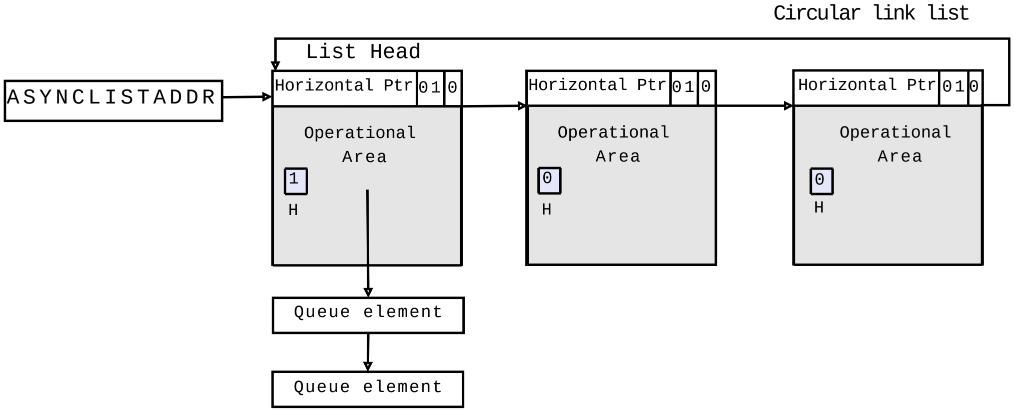

Figure 5.3: Asynchronous queue in host controller

Figure 5.3 gives an overview how asynchronous list looks like.
As shown in the figure the list always contains one queue head element which has *H* bit marked as 1. While traversing the list when controller hardware sees this bit set again, it knows that it has completed one iteration over the list. The host controller completes the processing of the list if one of the following events occur:

- End of the microframe occurs.
- Controller detects an empty list condition.
- The asynchronous schedule is disabled.

Node insertion

When inserting a new queue head into a activated asynchronous list software must ensure that schedule is always coherent from host controller's point of view. There should not be any invalid pointers in queue head as well as in linked queue elements. If queue head is the only element in the list then it should have head bit set. Actual logic for node insertion in the list is pretty much like insertion in a circular link list. See section 4.8.1 in EHCI documentation for details.

Node removal

Node removal from the list is more complex than insertion. The problem of node removal become complicated because of cached references to the removed node in the host controller hardware. Software must not remove any active nodes from the list. It should first mark them inactive and wait for hardware to reflect changes. Only then it should proceed to remove the inactive node.

A simple method is to disable the whole list and safely remove the marked node from the list. But this method has high overhead as for every removal driver has to disable whole active list.

A more elegant method called *handshake mechanism* ensures that controller hardware does not have any cached references to the removed node. This is a two step process. In the first step, the node to be removed is unlinked from the schedule by updating the linkage pointer but is not removed from the schedule. After unlinking from the schedule the driver sets *async advance doorbell* on host controller register. In step two, the driver will receive a notification (in form of an interrupt) that indicates that controller has traversed the whole list and now it does not have any cached pointers for nodes. At this point driver can remove the unlinked node from the schedule. Section 4.8.2 in EHCI documentation gives details about the node removal from the list.

5.3.4 PING Protocol Maintenance

USB 2.0 uses PING protocol on bulk OUT endpoints on high speed devices. The purpose of the protocol is to avoid unnecessary bandwidth usage. USB devices are slower than host controller hardware. While doing a transaction on an OUT bulk endpoint, device needs some time to stabilize and absorb the incoming data by writing it to persistent storage. It is possible that data has been transferred to device but controller did not receive any valid response from the device. This inefficient mechanism leads to bandwidth wastage. To overcome such scenarios USB 2.0 employs the PING protocol.

For every bulk out transaction the controller hardware first sends a PING protocol packet to the device. If a device is ready to accept more data it sends acknowledgment otherwise negative acknowledgment (NACK) back to the controller for the PING packet. If device sends a NACK for PING the host controller remains in PING state and postpones sending data to the next attempt. All bulk OUT data transactions end in PING state unlike IN transactions which end in complete state. For more details of states and PING protocol please refer to section 3.5 and 4.11 in EHCI and section 8.5.1

in USB 2.0 documents.

5.3.5 Data Toggle Synchronization

USB 2.0 uses data toggle protocol to ensure correct and orderly delivery of the data packets. The synchronization is achieved by using DATA0 and DATA1 type of data packets for delivery. The data toggle protocol works like stop-and-wait protocol with a window size of 1. After a successful reception of a data packet receiver toggles the packet bit. Similarly on the sender side when sender receives an acknowledgment for previously transmitted packet, it toggles its bit sequence. It is responsibility of the host controller driver to properly maintain these bits. The *data toggle* fields in queue elements and queue head will govern how these bit are manipulated. For bit field set to 1, DATA1 type packet will be used other wise DATA0.

The control transaction has specific requirements that command should always be DATA0 and status should be DATA1. The optional data stage is started with DATA1 and toggles with the number of packets. Similarly for bulk transactions after configuration and interface assignment the data toggle bits are set to zero. For every successful transaction the corresponding bit is toggled. Any mistake in data toggle maintenance can lead to obscure debugging situations which are hard to debug. If wrong data toggle bit is set into the transaction then host controller will not execute them and the system appears to be stopped, hence no other debugging techniques could help. Hence utmost care should be used when manipulating them.

Section 8.6 in USB 2.0 documentation gives details of the protocol and how synchronization is achieved over it.

Chapter 6

Distributed USB Infrastructure

6.1 Overview

In this chapter we will describe in detail about our implementation of distributed USB infrastructure for Barrelfish. There are four basic blocks in the implementation:

1. **USB Memory Manager:** This module is primarily responsible for memory maintenance. The explicit memory management is required to meet the complex memory requirements of USB protocol. It is responsible for allocating memory for control data, requests and I/O buffers.

2. **EHCI Host Controller Driver:** The EHCI host controller driver (HCD) is responsible for interaction with the host controller hardware. It abstracts and shields all complexities of managing EHCI hardware and provides a clean interface for communication with it.

3. **USB Manager:** USB manager is responsible for all management related activities for USB such as pipe allocation, keeping track of USB topology, performing device enumeration, locating device drivers for new device etc.

4. **USB Client Driver:** After device enumeration the responsibility of device is handed over to the USB device driver also called USB client driver. The driver knows how to operate and what services are available on the device.

The figure 6.1 shows the overall architecture of the implemented USB system. All modules, except USB memory manager, run in separate domains. The USB memory manager is compiled as a statically linked library

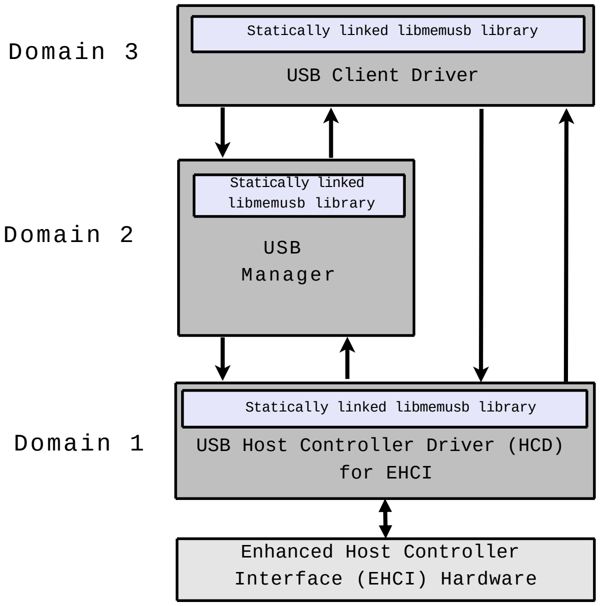
Domain 3
Statically linked libmemusb library
USB Client Driver
Domain 2
Statically linked libmemusb library
USB Manager
Domain 1
Statically linked libmemusb library
USB Host Controller Driver (HCD) for EHCI
Enhanced Host Controller Interface (EHCI) Hardware

Figure 6.1: Distributed USB infrastructure

to all three domains. The USB memory manager *can be implemented* as a separate domain but since all these module use it intensively for memory allocation it was decided to implemented it as a library. This design choice is good because every domain is responsible for managing its own memory usage. A notorious domain can crash USB memory manager by requesting a large chunk of memory which is first allocated on behalf of the memory manager. Crashing a single running instance of USB memory manager will effectively cease the execution of other domains also. Apart from unnecessary complexity of unmapping and mapping of capabilities across domains, USB memory manager, if implemented as separate domain, could also be a point of contention which is not desired from scalability perspective.

6.2 Communication Between the Domains

All main three modules run a server instance, to export services provided by them. The communication is done as in a client-server architecture by message passing. Messages are passed using inter-domain communication primitives provided by Barrelfish namely IDC and URPC. Each server instance first registers its services to system wide service registry server called *Chips*. All clients of a particular server manage to find it by requesting service name to Chips. These internal details of server client communication mechanism are hidden by Flounder, a domain specific language (DSL).

Flounder automatically generated the necessary back-end required to communicate with Chips and between a server and its clients.

6.3 Proxy Function Implementations

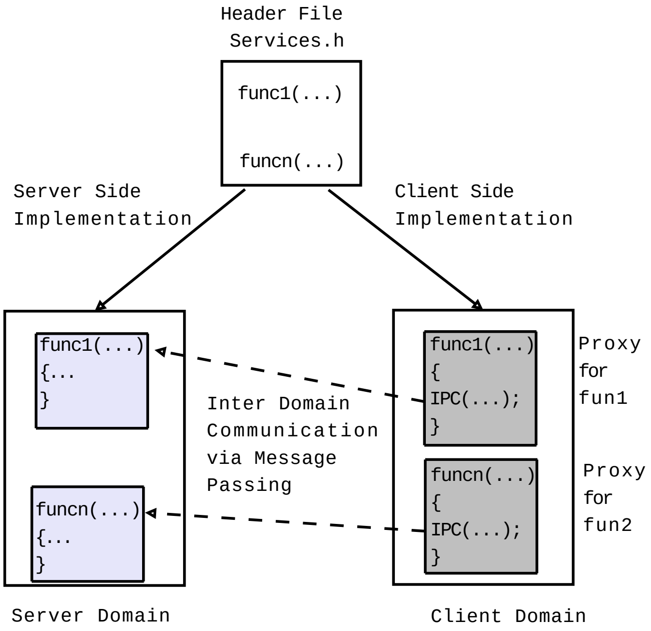

Figure 6.2: Proxy interfacing on client side

To shared relevant data structures and service functions, these pieces (prototypes) are moved to the header files. The definition of a particular service function depends on the domain. On server side a function is implemented in a normal manner. On the client side it is implemented as a proxy function. These proxy functions hide the internal server client communication mechanism from the top level logic. In every domain there are two layers of logic. The core logic is called *high level logic* which is the actual functionality implemented. The other, *lower level logic*, is responsible for taking care of communication and hiding other details. The figure 6.2 gives an overview of the implementation. The advantage of such design is that it completely decouples the communication logic from core logic of the implementation. It also improves readability because function names are same across the domains and new user can implicitly assume that control flow will *somehow* reach at the corresponding function in other domain. The inter domain communication is managed by the Flounder generated server client stubs.

6.4 Synchronization

There is no need for explicit synchronization on the server for multiple clients. For server client model implemented in Barrelfish every message is delivered one by one in a message handler loop. Next message is fetched only after processing for the current message is completed. This basic design choices make sure that there are no need for explicit locks in case of multiple clients sending requests concurrently to the server.

6.5 Hot-plugging

The implemented system is completely event driven. All participating modules wait for some activity or requests from higher layers. These modules just start as service servers. USB delivers activities from hardware as interrupts to the host controller driver. These interrupts drive further execution logic in the controller driver. Similarly other modules can be activated only upon demand. Upon no activity, these modules *can* be shutdown and unloaded (except for HC driver, which has interrupt handler required to get notifications on new activities) from the operating system. For example in current system implementation device driver starts as a server and waits for the *device found* notification. It could also be started only after when device is detected given that every executable contains same server name as its name.

Chapter 7

USB Memory Management

Barrelfish does not provide complex memory management operations to the user space applications. Each application is responsible for implementing such desired operations on a given basic set of capabilities by Barrelfish. These capabilities include, but not limited to:

1. Frame capability allocation. (assignments)

2. Mapping allocated frames to the virtual memory of applications. (map)

3. Revoking the capability containing frames. (delete)

The USB system implementation has complex memory management requirements. It allocates chunks of memory on the demand, pretty much like the malloc system in GNU Linux. Though USB specification does not put this as a requirement but in implemented system I/O buffers have to be page aligned. For all allocated queue heads and queue elements linkage is done in virtual space as well as physical space. The virtual space linkage is required to keep track of memory usage. Linkage at physical level facilitates host controller hardware to traverse and execute queue elements one after another. All relevant data structures have to be contiguous and 32 bytes aligned in physical space (pspace). Also USB memory manager has to ensure that no EHCI controller data structure crosses 4kB frame boundary.

The implemented system does not have to be as comprehensive as malloc utility. It already knows that all allocation requests are for either queue heads (48 bytes) or queue elements (32 bytes). Fortunately, all other potential request segments (specifically in device enumeration) such as device descriptors and command sequences also fit in less than that. For example USB device descriptor is 18 bytes long, which can easily fit in one 32 bytes chunk. The EHCI data structures need queue heads and elements to be 32 bytes aligned. Thus in the implementation the queue head size is rounded from 48 bytes to 64 bytes. This rounding up generates internal fragmentation of 16 bytes per allocated queue head.

The USB memory manager does memory allocation on demand. It supports chunk allocation of 32 and 64 bytes. For I/O buffers it allocates memory on page granularity. On initialization it allocates and maps few frames (2 frames to be specific) into the virtual space (vspace) of the process. The requests are allocated on these pages. On a typical x86_64 system page size is 4kB which equates to 128 and 64, queue elements (32 bytes) and queue heads (64 bytes) respectively. Memory management is done via link list of arrays. Every node in the link list contains one page worth of queue heads or queue elements as an array. USB memory manager maintains separate link lists for both of them. When a process consumes all free slots in a node, a new node is allocated with a new page and is inserted into the list.

The frame allocation is done via `frame_alloc` function call. Function takes size and capability reference structure. Upon successful allocation, the frame is mapped to vspace of process by `vspace_map_attr`. All such mappings created by USB memory manager are non cacheable. This is required because all such allocations are I/O memory region. They are readable and writable by hardware as well as by software. Software does the required setup on such regions in RAM and instructs hardware (here USB host controller) to execute the desired action. The result of execution is written back on these regions in RAM. For example arrival of data from disk, or setting flags by hardware to reflect execution status etc. Hence to get an up to date value of data from RAM these regions are marked non cacheable and every access to these region fetches data directly from RAM. The mapping flags contains `PTABLE_ACCESS_DEFAULT` and `PTABLE_CACHE_DISABLED`. To keep track of allocated frames memory manager maintains an internal data structure called `usb_page`. It contains virtual as well as physical address of the newly mapped frame with other relevant information. Every newly mapped page is entered into the link list of type `page_list_t` which is used to keep track of all allocated pages.

Memory allocation is done using `usb_mem` structure. Unlike malloc which returns address of newly allocated chunk (`void *`), the allocation in USB memory manager returns `usb_mem` type structure. The structure contains virtual address, corresponding physical address, size, type, valid and `capref` entry for cross reference purposes. Caller of allocation function has to recast the virtual memory according to the requirements. Size is size of the allocated block. Type can be one of the following:

1. Queue head with `USB_MEM_TYPE_QH`.
2. Queue element with `USB_MEM_TYPE_QE`.

3. I/O buffer with USB_MEM_TYPE_IO.

Rest of the fields in the structure are self explanatory.

7.1 Internal Memory Management

Internally USB memory management unit maintains three explicit link lists.

1. Link list 1 is named as queue head link list and identified as `struct qh_ll`. Allocation on this list is done in chunks of 64 bytes.

2. Link list 2 is named as queue element link list and identified as `struct qe_ll`. Allocation on this list is done in chunks of 32 bytes.

3. Third and the last list is called frame list. It is used to keep track and maintain the I/O buffers allocated. The granularity of list is one frame i.e. 4kB on typical x86_64 machines.

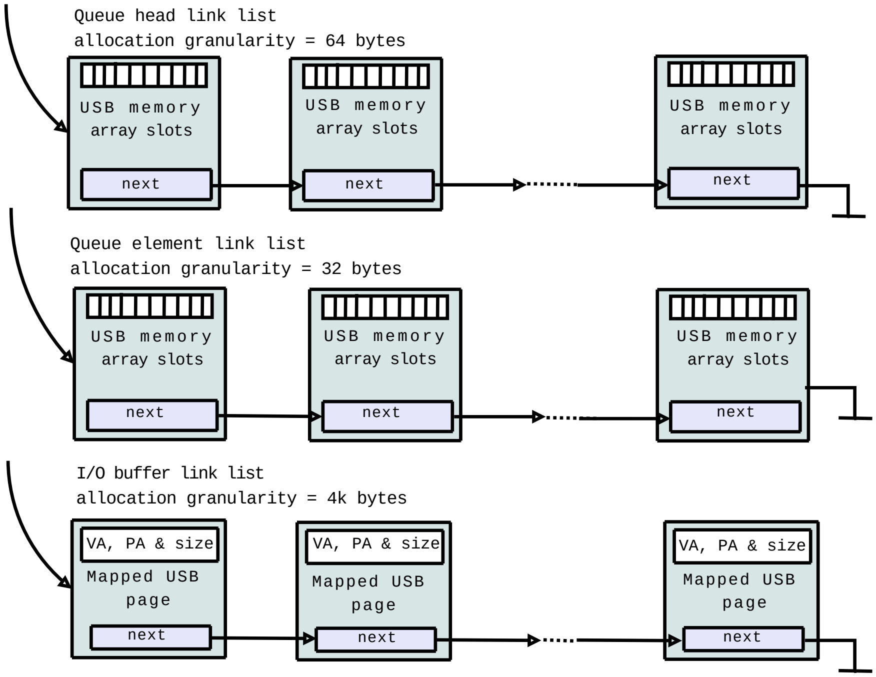

Figure 7.1: USB Memory Management

The figure 7.1 gives an overview of how whole system looks like.

For queue head and queue element link list the management is done similarly via link list of an array. Every node in the list contains following information apart from next linkage pointer:

- The corresponding usb_page entry.
- An array of usb_mem entries.
- Next free index in array.
- Total capacity, allocated and free slots in the node.
- An identifier for the debugging purpose.

For new memory allocation requests memory manager first tries to locate a node which has non zero free capacity. Before traversing the list it checks the global capacity of system if, that is zero then it immediately allocates a new page to the system. When the desired node is selected the allocation is done on it. The next free index on node tells which index in node's usb_mem array is free. The index is selected for allocation and it is updated to next free slot. The chosen slot is returned as allocated entry. The steps in allocation is same for both queue head and queue element but are done in different link lists.

Freeing up allocated memory is same. First the look up is done to find out the node on which allocation was done. This is done by matching the virtual address range of nodes. When node is identified the index is located by dividing the offset of usb_mem from node's virtual base address by the element size. Then index is marked free and next free index is updated to it.

I/O buffers are allocated on page granularity. Even for the small requests which are greater than 64 bytes but less than PAGE size a whole page is allocated. This is a simple but poor memory management. Ideally this should allocate just the desired amount. For typical for mass storage devices this is justified because they have block size of 512 bytes or 1 kB and reading them in a group will require buffer size of few kBs. I/O buffer allocation is done as described above for page allocation.

7.2 NUMA Aware I/O Buffer Allocation

NUMA stands for non-uniform memory access. The USB memory manager supports NUMA aware I/O buffer allocation. To initiate services caller first have to set EHCI core id on which the host controller driver is running. It can be extracted from HC driver using one of the offered services. In absence of valid core id, the passed flags are ignored by the allocator. The range of memory range is extracted from SKB. Upon requesting the SKB returns the memory range of the supplied core. Memory manager maintains this range for every core in the system. The data in this list is periodically refreshed to get more up to data from SKB. While allocating the I/O buffer, memory manager takes associated flags to do NUMA aware memory allocation. Three types of flags are supported:

- USB_NEAR_EHCI. This flag indicates to perform I/O buffer allocation on memory near to core on which HCD is running.
- USB_NEAR_SELF: This flag indicates to perform memory allocation near to core on which driver is running.
- USB_DONT_CARE: This flag indicates to use whole range of memory.

These flags are guidelines rather than rules to do memory allocation. If memory manager has not sufficient data to do allocation it will ignore them. Upon fixing the memory range the `ram_set_affinity` function is called to set range before doing the frame allocation for the request.

7.3 Library Interface and Functions

Since all other three modules, namely host controller driver, USB manager and USB device driver, need functionality of USB memory manager it is linked as a static library to them. Following functions are provided by the `libmemusb` library

- `void usb_mem_init(void):` This is an initialization function to the memusb library. Upon invoking it allocates and initializes the both link lists with 1 page each.
- `usb_mem malloc_qe(void):` This function is called to allocate a chunk of USB memory of worth holding a queue element i.e. 32 bytes.
- `usb_mem malloc_qh(void):` This function is called to allocate a chunk of USB memory of worth holding a queue head i.e. 64 bytes.
- `usb_mem malloc_32(void):` This is a wrapper function which internally calls `malloc_qe` for allocation. This is used for readability purposes. All allocations in non-HCD domains are done via this.
- `usb_mem malloc_64(void):` Similar purpose function as explained above to allocate 64 bytes worth of USB memory.
- `usb_mem* malloc_qe_n(int n):` This function allocates an array of n elements of 32 bytes each.
- `usb_mem* malloc_qh_n(int n):` Similar function to allocate an array of n element of 64 bytes each.

- `void free_qe(usb_mem mem):` This function is free equivalent of malloc utility which allocates 32 bytes worth of segment. Upon receiving an invalid mem element it simply ignores the call to free the mem element.

- `void free_qh(usb_mem mem):` Similar functional equivalent for queue head allocations.

- `void free_32(usb_mem mem):` Wrapper utility function for better readability which internally calls `free_qe`.

- `void free_64(usb_mem mem):` Wrapper utility function for better readability which internally calls `free_qh`.

- `usb_mem malloc_iobuff(int sz, int NUMA_FLAG):` This function allocates I/O buffer of size big enough to hold sz bytes worth of data. The number of pages required are calculated and are allocated and mapped to the process's vspace. The NUMA_FLAG is used to do NUMA aware allocation of request.

- `void free_iobuff(usb_mem mem):` Equivalent free function to free up the buffer for future use.

- `void* map_cap(struct capref, uint32_t sz):` This function is used to map sz worth of bytes of a given cap to the process's vspace.

- `void print_usb_mem(usb_mem mem):` Debugging utility function to print out `usb_mem` type structures.

- `void print_memory_stats(void):` Debugging utility function to print out memory statistics about the current status of USB memory. It prints out data about how many frames are in use, how many 32 and 64 bytes segments are allocated, freed and what is current capacity. It helps in tracking memory leaks. Every allocation should have corresponding free.

Chapter 8

Host Controller Driver (HCD)

The USB host controller driver is responsible for managing the host controller hardware. The implemented driver is for Enhanced Host Controller Interface (EHCI) controller. The chapter 5 gives more comprehensive overview of the host controller hardware and responsibilities. In this chapter we will focus on software interface and services provided by the implemented host controller driver (HCD).

8.1 PCI Interfacing and Booting the Host Controller

The primary responsibility of HCD is to manage host controller hardware. EHCI controller hardware resides on PCI bus, hence the first step towards starting the driver is to detect device and start associated services.

The stand alone host controller driver starts with initializing USB memory initialization logic. It then proceeds to initiates the low level server client interfaces and registers the services to the Chips service server. This much work is done regardless of whether PCI finds a host controller or not. It also establishes a client relationship with the USB manager server.

In Barrelfish PCI funtionalities are also implemented as a server (PCI server). The HCD then connects to the PCI server and probe for EHCI controller hardware. If controller is found (which is typical on every modern day system[1]), PCI server notifies the HCD by a PCI callback function. The callback function contains information for memory mapped PCI region where EHCI controller registers are mapped. The current implementation

[1] Qemu does not support EHCI as of with version 0.9.1.

does not support 64 bit controllers so upon checking the relevant flag for controller's capability, memory affinity is set to 4 GB, which is accessible from 32 bit controllers.

The HCD uses Mackerel interfacing to interact with host controller hardware. The memory region, where HC registers are, is mapped into HC driver's vspace before actually booting the hardware. Mackerel generated code vastly reduces the interaction complexity with controller hardware registers and provides a very clean, efficient interface to write and read them. Since capability registers set can be of variable length, the implementation uses two different HCD devices to represents two different registers sets. The `ehci_cap.dev` file contains capability registers set device information. Similarly `ehci_op.dev` file contains operational registers set device information. First EHCI capability device is initialized and offset of operational set is extracted from there. In second step EHCI operation device is initialized.

To orderly initialize the controller hardware, the implementation follows the 5 steps as outlined in Intel EHCI documentation ([32]). These are:

1. Specify the 4GB segment where the interface data structure are allocated. For 32 bit controller this value effectively will reduce to `00000000h`.
2. Set the interrupt map and enable the interrupts generation. For the current implementation following interrupts are enabled:
    - On async advancement.
    - On grave host controller errors, on which software intervention is required.
    - On port status change, when a device is inserted or removed.
    - On transaction completion (IOC).
3. Write periodic frame base address and if required then enable the schedule. In the current implementation this is disabled.
4. Set the interrupt threshold, frame list size etc. and set controller ON via run/stop bit.
5. Enable global routing to route all interrupts to EHCI. Since we are not targeting the backwards compatibility all interrupts should route to EHCI. EHCI is capable of handling devices operating on USB 2.0 standards. OHCI and UHCI are used for USB 1.0 and USB 1.1 standards respectively.

At this point controller hardware is configured, up and running.

8.2 Event Notifications

The controller hardware generates interrupts on new events. There interrupts are delivered to HCD as IDC messages. Upon receiving any such notification the interrupt handler function is invoked. The handler function scans the EHCI status and interrupt registers to locate the source of the interrupt. It then flags appropriate threads to do the interrupt processing job. The foot print of interrupt handler should be small to avoid excessive processing while handling an interrupt. Doing large amount of work in interrupt handler may interfere with the responsiveness of the system. HCD knows the destination for each event and proceeds accordingly. Port status change interrupts are forwarded to USB manager as inter domain messages. Others like transaction complete interrupts are processed locally in HCD but in different threads.

8.3 Device Detection and Port Power

When there is any activity on USB ports, HCD receives an interrupt about port status change. Here activity can correspond to device attachment or removal. In both cases, upon receiving the interrupt first port of activity is located by linear scan of all the ports. Port with activity has *connect status change* bit set in associated port register (PORTSC). Upon locating the port the *current connect status* bit is checked. If bit is set then a new device has been inserted otherwise device which was present there is removed.

When a new device is located HCD then proceeds to port reset and power logic. The *line status* tells the HCD that whether device is a low speed device or not. Next the port with device present is reset and enabled. At this moment HCD notifies the USB manager that a new device has been located to a specific port number. In a similar way upon device removal USB manager is notified and USB manager is responsible for proper clean up.

8.4 Asynchronous Queue Management

The primary responsibility of HCD is to manage asynchronous queue. Asynchronous queue elements include control, bulk and interrupt requests. For a particular type of request a queue head and multiple queue elements are allocated. HCD has to keep track of all requests en-queued, completed or aborted. HCD also has to allocate and free associated resources with the requests.

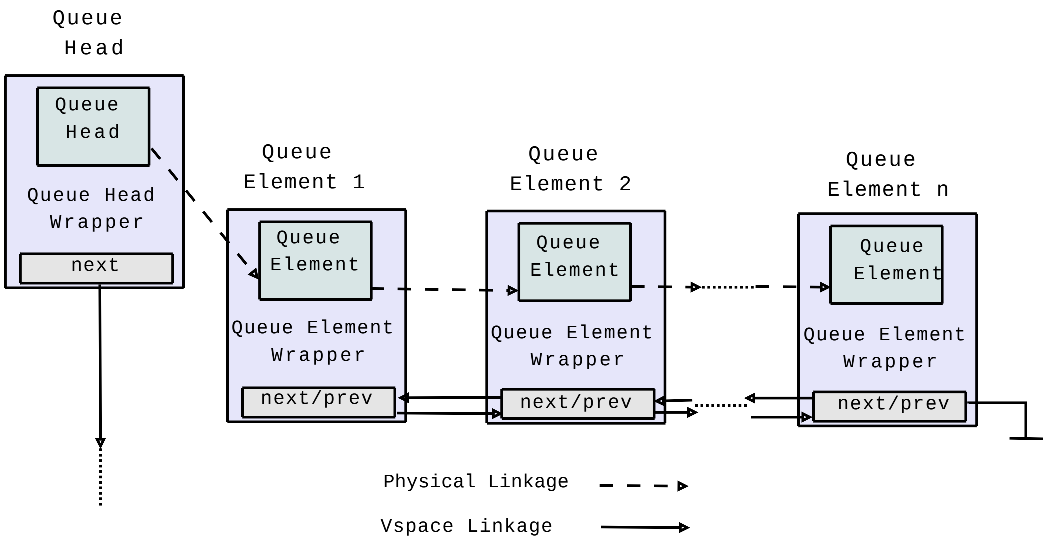

Figure 8.1: HCD asynchronous queue management overview

EHCI controller traverse physical linkage of these queue elements to fetch next executable element. On the other hand to keep track of these elements HCD has to keep track of them in vspace. These requirements lead to queue management design as shown in figure 8.1.

The queue management is done on two level, virtual and physical. Physical linkage is done by putting higher 27 bits of physical address of next queue element in next pointer of queue element data structure. The higher 27 bits are required as all queue elements are 32 bytes aligned. The vspace linkage is done with queue element/head wrappers. These structures contain required meta data and information for proper queue management apart from the queue element/head references.

The queue head wrapper contains following information:

- Reference to the actual queue head element.
- Pointers for maintaining doubly link list.
- Service reference to the client from where this request has come.
- References to the queue element (qTD) list wrappers.
- Status flag information.
- Total data length expected for this transaction.
- Debugging stamp.

Every allocated queue head is wrapped into this structure and is then passed to internal queuing mechanism.
Similarly the qTD wrappers also contain following information:

- Reference to the actual queue element.
- Pointers for link maintenance.
- Value of `usb_mem`.
- Reference to queue head wrapper.

When a new request is received in HCD, the HCD first allocated the necessary numbers of queue elements and links them in vspace and pspace. Vspace linkage is done using wrappers. After then, they are passed to to link with allocated queue head. When whole list in linked then the reference to the queue head is passed for insertion in asynchronous schedule.

8.5 Transaction Management

After all necessary data elements are allocated they are passed to internal queue manager. The internal queue manager is responsible for inserting them into the host controller's hardware queue. The last queue element in the the queue has interrupt on completion bit (IOC) set to one. Hence transaction can be considered complete if device or host[2] transfer a short packet or last queue element is executed. In both cases it will result in an IOC.

EHCI controller has very complex queue management logic. It is very important to adhere to the guidelines given in EHCI documentation otherwise bugs generated from these are *very hard* to debug. As explained in chapter 5, EHCI controller maintains circular link list of queue heads linked on physical pointers. The 1 bit flag *head* in queue head is responsible for telling hardware that one complete traversal of list has been done. There is only one queue head in whole hardware queue that has this bit set to 1. Due to caching issues by controller the removal of a node from the hardware queue is done in two steps. First unlinking then removal from the asynchronous schedule. This is called *handshake mechanism* explained in detailed in chapter 5.

The hardware queue is managed by two separate threads. These two threads loosely share the responsibilities of two step handshake mechanism. These threads are started at the boot time when internal queue manager

[2] There are no cases when host should do this but USB devices certainly can for IN transactions.

is initialized. Upon receiving an IOC the interrupt handler signals the first thread to scan the hardware queue and locate which queue heads have completed the transactions. IOCs in EHCI are cumulative and their frequency depends upon interrupt threshold. In current implementation we use default which is 1 millisecond. Hence for a one IOC, there might be more than one queue heads which have finished processing. The first thread unlink them from hardware queue but does not de-allocate them because controller might have cached references to them. If any one of these heads have *head* bit set to 1, other queue head is chosen to set its *head* bit to 1. After complete processing, the first thread set the doorbell for async advancement on controller. At this stage queue heads are removed from hardware queue but they are still linked in vspace via wrappers and are not de-allocated.

Upon receiving the interrupt on async advancement, the interrupt handler signals the second thread to remove and free the marked queue heads from the asynchronous schedule. The async advancement interrupt implies that controller now does not have any old cacheable pointers which might lead to removed nodes. The second thread scans and remove all removable nodes from the list. Wrapper for every node removed contains identification for the client process. For every node, number of bytes transferred is calculated if transfer was successful otherwise error number is generated. For every node removed an IDC is send to corresponding process to indicate that transfer has been completed. Upon detecting that there are no more nodes in asynchronous schedule, the schedule is disabled.

A new node insertion in schedule is quite simpler compared to node removal. The list is checked if it is NULL. If list is found to be NULL to be inserted node is the only one. The head bit is set and node is inserted into the schedule. On the other hand if schedule is found not-NULL, node is inserted as done in link lists. The linkage is done on both vspace as well as pspace.

8.6 Host Controller Server's Services

The host controller driver acts as server as well as client to other domains. At the time of booting HCD registers its services to the Chips service server. Then it proceeds to connect as client to the USB manager server. USB manager follows the same procedure. In the end both HCD and USB manager are servers as well as clients of each other.

The following services are provided by EHCI HCD server:

1. **Map shared state:** Some crucial device states such as data tog-

gle state[3], device connectivity status etc. are shared between HCD and USB manager. USB manager is responsible for maintaining the USB topology tree. HCD only notifies USB manager about device removal or attachment. Similarly, data toggle status is initialized by USB manager but is used by HCD. HCD does not know when the data toggle bits are re-initialized e.g. on re-configuration of the USB device. These are frequently used data structures by HCD but their maintenance responsibility and semantics lies in USB manager. Hence they are shared between these two. After the connection between HCD and USB manager is established, USB manager calls `map_dev_arr` function to pass capability of shared page. HCD then maps this capability into its vspace and use it. For current implementation this is one frame but size can expand as required number of items to be shared increases.

Function call: `void map_dev_arr(struct capref cap, uint32_t sz);`

2. **Execute data-less control request:** As explained before control sequences can have optional data stage. Example of data-less control sequences being, assignment of address, configuration and interface to the USB device etc. In data-less control sequence, transaction is done with only two stages, command and status. The client side implementation of service function is responsible for marshalling the arguments for the call. The explicit marshalling is required because USB command sequence is exactly 8 bytes. In normal structure definition in C, compiler is free to pad the size of structure to make it word or dword aligned. Hence explicit marshalling of parameters are done, when passing around in inter domain communication. The function takes three arguments, USB command request, device id and debug flag.

   Function call:
   ```
   int usb_ctrl_exe(usb_device_request usb_req,
                               uint8_t device, int debug);
   ```

3. **Execute with-data control request:** Most of the control sequences have associated optional data stage with them. In 3-stage control transaction command sequence is followed by data stage and then status stage. In data control sequence apart from the command sequence the caller also have to provide physical address of the data buffer and expected data size. USB allows to terminate the data transaction with less number of bytes expected but it does not allow to overflow it. In

[3] Explained in chapter 5.

case of overflow it results in data babble error. The HCD takes care of allocating space for command and status sequences which are 32 bytes each and linking them. Upon execution termination, the HCD does downward calls to client side to notify that execution has been done. If execution has been successful HCD returns number of bytes received or send, otherwise error code. All control sequence transactions assume that it is done on default endpoint (0), hence there is no need to pass it explicitly.

Function call:
```
int usb_dctrl_exe(usb_device_request usb_req,
                  void* buff, uint64_t sz,
                  uint8_t device, int debug);
```

4. **Execute bulk transaction request:** The bulk transaction requests are done on stream pipes (unlike control sequences which are on message pipes). The bulk transactions are done on bulk endpoints. Contrary to default end point, all other USB endpoints are uni-directional. USB bulk transaction is similar to control transaction in terms of setup but it does not have explicit command, data and status stages. The only purpose of bulk transaction is to transfer certain amount of bytes from device to RAM or vice-versa. Queue head allocation, queue element linking is done similarly as done in command sequences. Number of queue elements depends upon the data size. One queue element can hold up to 20 kB worth of data (with proper page alignment, which is implemented in the system). The queue elements are allocated by HCD and are freed upon completion of the transaction. A transaction can result in success or error. Upon successful completion the number of bytes moved is reported back to the client, otherwise error code is propagated.

   Function call:
   ```
   int usb_bulk_exe(usb_pipe_t pipe, void* io_buff,
                    uint32_t len, int debug);
   ```

5. **Get HCD core id:** This is a simple function which returns on which core HCD is running. It is required by other module's `libmemusb` to perform NUMA aware memory allocation.

   Function call: `int get_ehci_core_id(void);`

Chapter 9

USB Manager

The USB manager is responsible for maintaining USB infrastructure related services. Similar to HCD, the manager is started as a server. In the beginning the communication between USB manager and HCD has to be synchronized. These both modules are servers and clients of each other. The primary responsibilities of USB manager include device enumeration, USB tree topology maintenance, allocation of configuration and interface to the device, allocating pipes to device drivers, maintaining USB device's meta-data etc.

9.1 Device Plug & Unplug

When a new device is inserted into one of the USB ports (also with device removal), HCD sends notification to USB manager. USB manager is responsible for taking appropriate actions.

Upon receiving device attachment notification, the USB manager proceeds to USB device enumeration. After successful enumeration and address assignment, it looks for driver for the device by asking SKB. When an appropriate driver is located an entry for the device is made into the USB tree topology and control is handed over to the device driver.

If a device removal notification is received, USB manager immediately sets the connectivity status as disconnected and then proceeds to remove all other devices which might be connected by this device[1]. HCD use this shared connectivity information to accept or reject transactions from device drivers. USB manager then notifies device driver that the device has been removed.

[1] USB hubs are also treated as USB devices.

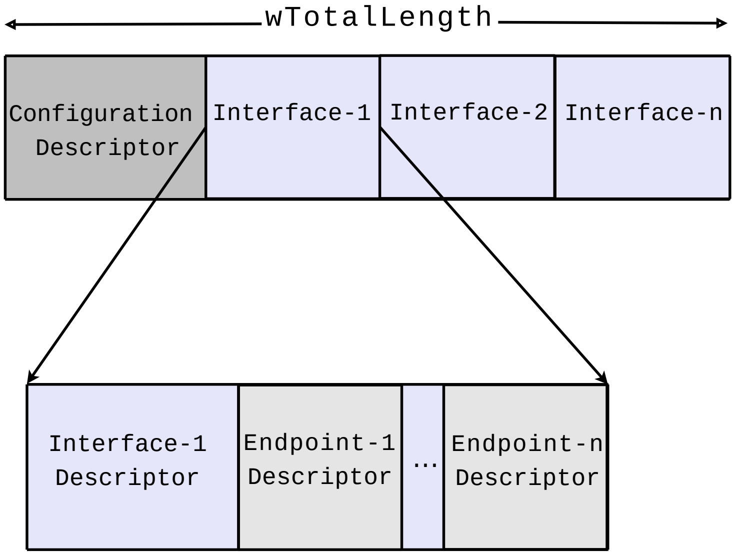

Figure 9.1: Configuration descriptor with interface and endpoint descriptors

9.2 Device Enumeration

The USB manager maintains detailed information containing device descriptor, configuration descriptor, interface descriptor, endpoints, string descriptors etc, port, address etc. for every enumerated device. When a new device is found following steps are taken by enumeration thread:

1. A new slot is located into the USB device array. Since USB 2.0 can only support 127 devices, a static array with 127 potential entries is created. The index into the array is used as address for new USB device. Address zero is reserved and is only used at the enumeration time before device is assigned a proper non-zero address.

2. USB manager then tries to read device descriptor. Device descriptor contains information regarding class, subclass, protocol, string descriptors indexes and the number of configurations etc. The vital information here is number of configurations. The device descriptor is saved into the USB device structure.

3. USB manager reads every configuration reported by device descriptor. The configuration contains information regarding power requirements, interfaces and endpoints numbers. The configuration is actually read twice. During the first reading only configuration is read to find out total length of data needed to read interface and endpoint descriptors as well. This is reported via *wTotalLength* field. During second reading configuration is read again but with potential data length as

wTotalLength instead of just configuration size. The total data layout is shown in figure 9.1.

Every descriptor has its associated length in *bLength* field. Hence during second read USB manager gets the whole data and initializes all interfaces and endpoints descriptors and saves them into the device structure.

4. At this step USB manager have all the required information to address and configure the device. Now USB manager assigns a proper non-zero address to the device and cross checks by reading device descriptor again but from newly assigned address. USB 2.0 documentation does not specify the exact order when address can be assign to the newly found device.

5. Now USB manager assigns a particular configuration and interfaces to the device. During the current implementation the flash mass storage device only reported one configuration and one interface descriptor, hence USB manager does not have any choices. But in general it should consult SKB to choose proper configuration. These configuration and interface can later also be changed by device drivers.

6. After step 5, the USB device is in addressed mode, with configuration and interface assigned. Now USB manager proceeds to locate appropriate device drivers for the device by asking SKB. If a match is found, SKB returns the server name which is running associated device driver.

7. USB manager tries to communicate with the driver server and probe the server if it accepts new device or not. If device is accepted by the server, an entry for the device is created into the USB topology tree. Otherwise if no appropriate drivers are found, the associated slot in device array with all other resources are released.

9.3 USB Tree Topology

The USB manager is responsible for maintaining the USB topology tree. From the root hub perspective every port seems directly connected to it. If a hub or device (device may be a compound device supporting multiple device interfaces) is disconnected from the port then USB manager is responsible for maintaining the consistent view of USB topology. This topology tree is used to update the connectivity status which is shared between USB manager and HCD. Figure 9.2 gives an overview of maintained USB topology tree.

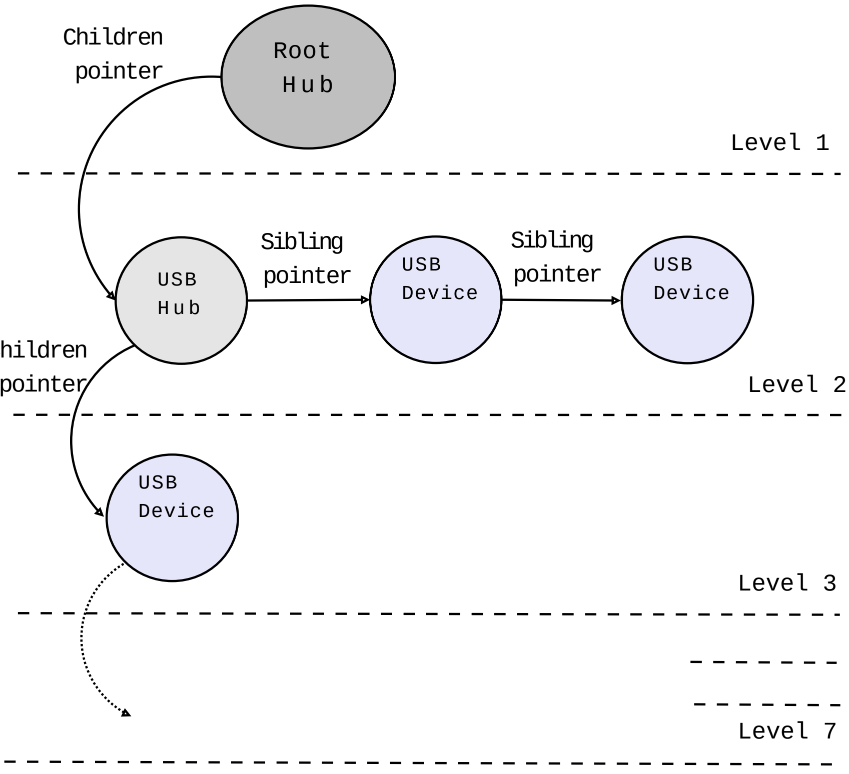

Figure 9.2: USB tree topology as maintained by USB manager

Initially USB topology tree contains a single node representing USB root hub. When a new device is attached directly to the root hub, the associated entry is made into the tree. If a device is attached via already connected USB hub then associated hub entry is passed as its parent. For devices connected to root hub, parent is passed as NULL. When a new device is passed for insertion, first the parent node is located into the tree. USB 2.0 does not allow depth more than 7 levels. Hence no device below depth 7 is inserted. Upon locating the parent node, new node representing new device is inserted into the children list of the parent.

When a device is removed HCD notifies the USB manager. USB manager locates the node containing the device address. It then proceeds recursively to remove all its children and their sub-trees. During this process when a device node is removed corresponding to it status is also updated into the shared array as disconnected from root hub and corresponding address slot is marked free.

9.4 Communication

USB manager has a very complex communication logic. Not only it acts as a server to HCD and USB device drivers, it is also client of both. As a server, it provides different services to HCD and device drivers. There is no common function which is required by both. Next few sections gives an overview of it.

9.4.1 With HCD

HCD and USB manager have a very critical server client communication setup. Both are servers as well as clients of each other. They both connect and synchronize at the booting time. The implicit steps followed are:

1. USB manager starts partial server by explicit message handling [2], until HCD is connected.
2. When connection from HCD is confirmed, USB manager connects to HCD server. HCD server does same thing as did by USB manager in previous step.
3. When both are connected to each other as a client, both proceed to do initialization of internal logic and enter `messages_handler_loop`.

The synchronization between servers and clients becomes tricky because, right now, Barrelfish does not support threaded message handling. There *should* only be one message handler loop in whole process. Hence these explicit wait-connect-start steps are required.

HCD and USB manager communication when a new device is inserted or an old one is removed. More details about these services are given in service section.

9.4.2 With Client Driver

Like HCD, USB manager also connects as a client and as a server to the client driver. During the device enumeration process, when a device driver is located for a newly connected device, USB manager connects as a client to the device server. USB manager then passes the required information to `probe` function on server side. If driver is ready to accept the device then it will return `ACCEPT` or `REJECT` otherwise. This is a one time communication and USB manager can close client connection to the device server after this step.

[2] i.e. not calling messages_handler_loop(), which is a non-returning function.

USB device drivers need certain information about the device from USB manager like availability of particular kind of pipe, configuration and interface information etc. Hence they connect to USB manager as a client. These service are provided to client driver and details are outlined in service section. Services provided to client driver are different from HCD.

9.4.3 With SKB

Unlike other two communication interface, communication with SKB is mostly one way. USB manager is responsible for adding facts and system updates to SKB. SKB is responsible for reflecting them to the other modules in the system. For every device following facts are added to the SKB

1. Port number on which it is connected.
2. Assigned device address.
3. Maximum power, if device is not self-powered.
4. Class, subclass, protocol codes.
5. Enabled configuration and interface number.

SKB also helps USB manager in locating a device driver for new devices. On booting time device drivers register *interest information* to SKB. USB manager search this to match and locate driver server.

9.5 USB Manager Server's Services

In current implementation USB manager provides following services to HCD and device drivers.

9.5.1 To HCD

1. **Device connect notification:** When a new device is connected, HCD after port reset and power calls device connect notification function to notify USB manager about new device.

   Function call: `void notify_new_device(int port_num);`

2. **Device disconnect notification:** When a device is unplugged, HCD notifies USB manager about removal of the device.

   Function call: `void notify_device_removal(int port_num);`

9.5.2 To Client Driver

1. **Locate pipe:** Pipe is a logical connection between a device driver and data source/sink on the device. The USB driver expects some specific type of endpoints to be present on the device in order to work properly. When device responsibility is handed over to the driver after probe function, device driver requests look-up for specific pipe types to the USB manager. USB manager after receiving such request looks up in the device structure and initializes the pipe and returns it to device driver. A valid bit in pipe structure tells if request was successful or not. A pipe is required for all transactions done on non-control type pipe. All control transactions are done on default pipe, hence there is no need for an explicit pipe for endpoint zero. A pipe structure contains following information:

   - Device address.
   - Endpoint number.
   - Endpoint address.
   - Endpoint direction (IN or OUT).
   - Endpoint type (control, bulk, interrupt or isochronous).
   - Maximum packet size. Though USB standardize packet size for different endpoints, like 64 bytes for control, 512 for high speed bulk points but few non-complaint devices do not follow these regulation. Hence it is recommended to extract this information from the endpoint itself.
   - Multiplier for high speed isochronous points.

   Function call:
   ```
   void init_pipe(uint8_t dev, uint8_t type,
                  uint8_t dir, usb_pipe_t *req_p);
   ```

2. **Current configuration:** The function returns the current active configuration on the device to the driver. Driver may parse it to locate other capabilities on the device. Return value indicates the success or failure of the request.

   Function call:
   ```
   int get_curr_config(uint8_t dev,
              usb_configuration_descriptor *config);
   ```

3. **Get number of configurations:** This function returns the number of available configurations on the specified device.

   Function call:
   ```
   void get_num_config(uint8_t dev, uint8_t *num);
   ```

4. **Get specified configuration:** This function returns the specific configuration descriptor to the caller. The return value indicates the status of the request.

   Function call:
   ```
   int get_config(uint8_t dev, uint8_t num,
       usb_config_descriptor *config);
   ```

5. **Set configuration:** This function sets the specified configuration on the device and returns the status of the request.

   Function call:
   ```
   int set_config(uint8_t dev, uint8_t num);
   ```

6. **Current interface:** The function returns the current active interface on the device to the driver. Driver may parse it to locate other capabilities on the device. Return value indicates the success or failure of the request.

   Function call:
   ```
   int get_curr_intf(uint8_t dev,
       usb_interface_descriptor *intf);
   ```

7. **Get number of interfaces:** This function returns the number of available interfaces on the specified device.

   Function call:
   ```
   void get_num_intf(uint8_t dev, uint8_t *num);
   ```

8. **Get specified interface:** This function returns the specific interface descriptor to the caller. The return value indicates the status of the request.

   Function call:
   ```
   int get_intf(uint8_t dev, uint8_t num,
       usb_interface_descriptor *intf);
   ```

9. **Set configuration:** This function sets the specified interface on the device and returns the status of the request.

   Function call:
   ```
   int set_intf(uint8_t dev, uint8_t num);
   ```

Chapter 10

USB Device Drivers

USB device drivers or client drivers are very much like normal device drivers except instead of directly communicating to a device (such as writing registers) they interact with devices over the USB protocol. USB does not interpret the communication over its stream pipes. The drivers sits on top of USB stack and knows what services are available on current the device. USB device drivers are responsible for implementing services on top of USB protocol, such as receiving or sending data to the device etc. Different type of devices (bulk storage, human-interaction devices etc.) have different protocol specifications which run over USB. In this chapter we will talk in detail about USB mass storage protocol [33] and the current implementation in the system.

10.1 Mass Storage Protocol Specification

A mass storage device is an electronic device which can store information on persistent storage and supports data transfer on a hardware interface. The data can be anything ranging from executable files to databases, media files, spreadsheets etc. USB is used intensively for variety of storage devices. Devices such as magnetic hard drives, optical drives, flash memory devices, digital cameras, mobile devices etc. can easily connect over USB to perform data transfer. The USB mass storage specification does not provide any particular file-system specific services. Instead it provides a very simple set of services like read, write, verify data blocks on the device. It is the responsibility of higher layers such as file system to interpret these data blocks.

The table 10.1 gives overview of mass storage class interface descriptor. The interface class and protocol code for mass storage are 08h and 50h respectively. The subclass code specifies which type of device is it. A typical flash drive uses SCSI transparent command set (06h).

Table 10.1: Bulk-Only data interface descriptor

Offset	Field	Size	Value	Description
0	bLength	1 byte	09h	Size of this descriptor in bytes.
1	bDescriptorType	1 byte	04h	INTERFACE descriptor type.
2	bInterfaceNumber	1 byte	0?h	Number of interface.
3	bAlternateSetting	1 byte	??h	Value used to select alternate interface.
4	bNumEndpoints	1 byte	??h	Number of endpoints used by this interface excluding endpoint zero. The value shall be at least 2.
5	bInterfaceClass	1 byte	08h	MASS STORAGE class.
6	bInterfaceSubClass	1 byte	0?h	Sub class code. Indicates which industry standard command block definition to use.
7	bInterfaceProtocol	1 byte	50h	BULK-ONLY TRANSPORT.
8	iInterface	1 byte	??h	Index to the string descriptor.

The USB bulk-only mass storage protocol works on command-data-status flow. The USB host sends command block, the host or device can send data and then the device returns status. All reading and writing is done on logical blocks. A driver can assume them to be like a long array. USB transactions does not know about these details. Device firmware, which implements USB mass storage protocol, is responsible for interpreting them. Like USB control transactions (but completely unrelated), the bulk transfer over USB has three phases, command, optional data stage and status. In command phase host controller send command block to the device OUT endpoint. It is then followed by optional data stage where requested data in command phase is transferred. At last, status block is returned by device's IN endpoint. The status reports the status of transaction. It is quite possible that data has been transferred but the command failed because either the device was busy and did not save the data or its internal buffer overflowed. For commands which do not have data stage, HC immediately proceeds to read status. USB does not specify how much longer the HC should wait before reading status. The figure 10.1 shows how protocol runs.

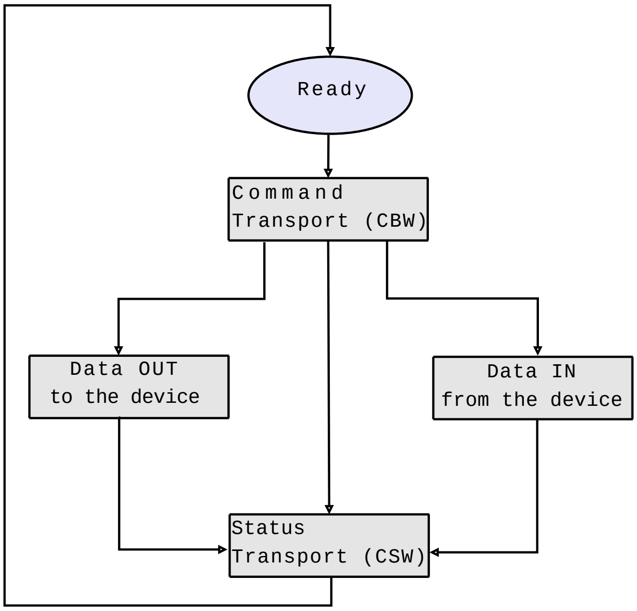

Figure 10.1: Command/Data/Status Flow

10.1.1 Command Block Wrapper (CBW)

The command block is transported in command block wrapper (CBW). CBW is exactly 31 bytes. The size is important because GCC pads structure to 32 bytes for alignment purpose. Sending a 32 byte structure to the device may results in unnecessary stall of endpoint, which is hard to debug. All USB devices follow little endian formatting hence LSB (byte 0) is transferred first to the device.

Table 10.2: Command block wrapper

	Bit							
	7	6	5	4	3	2	1	0
Byte 0-3	dCBWSignature							
Byte 4-7	dCBWTag							
Byte 8-11	dCBWDataTransferLength							
Byte 12	bmCBWFlags							
Byte 13	Rsvd(0)				bCBWLUN			
Byte 14	Rsvd(0)			bCBWCBLength				
Byte 15-30	CBWCB							

The table 10.2 gives an overview of command block wrapper looks like. It contains following fields:

- **dCBWSignature:** The signature helps to identify this block as a CBW. It always contains value of 43425335h.

- **dCBWTag:** The tag is generated and set by the mass storage driver. It is used to identify and match corresponding status packet. The device echoes the tag value in returned status block.

- **DCBWDataTransferLength:** The number of bytes that the host expects to transfer during data stage. The direction is obtained from bmCBWFlags field. If transfer length is zero then there is no data stage.

- **bmCBWFlags:** Following information is saved in the bitmap
  Bit 7 : Direction of transfer. 0= Data out, 1= Data in.
  Bit 6 : Obsolete.
  Bit 5-0: Reserved.

- **bCBWLUN:** It contains device's logical unit number (LUN). Device supporting more than 1 LUNs are rare. Number of LUNs are reported in the INQUIRY command[1]. For device supporting only 1 LUN, this should be set to zero.

- **bCBWCBLength:** It contains length of the SCSI command to be transferred. The legal values are 1 to 16 bytes.

- **CBWCB:** It contains actual SCSI command to be executed by the device.

10.1.2 Status Block Wrapper (CBW)

Similar to CBW, status also have status block wrapper. The table 10.3 gives an overview of the structure.

Table 10.3: Status block wrapper

	Bit							
	7	6	5	4	3	2	1	0
Byte 0-3	dCSWSignature							
Byte 4-7	dCSWTag							
Byte 8-11	dCBWDataResidue							
Byte 12	bCSWStatus							

The CSW is exact 13 bytes. It contains following fields:

[1] Explained later.

- **dCSWSignature:** The signature that helps in identify this data packet as a status block wrapper. It always contains 53425355h.
- **dCSWTag:** The device always sets this field to the corresponding value received in the associated command packet.
- **dCSWDataResidue:** In this field, the device reports the difference between the amount of data expected as stated in command block and the actual data processed by the device.
- **bCSWStatus:** The value indicates the success or failure of the command. Valid values are shown in table 10.4.

Table 10.4: Valid command block status values

Value	**Description**
00h	Command passed.
01h	Command failed.
02h	Phase error.
03h and 04h	Reserved (obsolete).
05h to FFh	Reserved.

For detailed description of protocol please refer to *Universal Serial Bus, Mass Storage Class, Bulk-Only Transport* documentation [33].

10.2 Mass Storage Class Driver

Like other modules of USB infrastructure, USB mass storage driver also starts by registering and starting the device server. It also registers the USB device of interest to the SKB.

The device control is transferred to the driver[2] by probe function. In probe function USB manager passes device related information and inquires the driver. The driver upon accepting the device sends notification to USB manager. When driver accepts the device, it immediately connects to USB manager and HCD as a client.

The primary responsibility of the driver is to read meta-data and configure the device to read/write logical blocks. The implemented driver takes following steps:

[2] Driver here on only refers to implemented mass storage driver

1. For a mass storage device to work properly it is required to have at least one IN and one OUT endpoints. Mass storage protocol can use endpoint zero as a control endpoint. Hence the driver calls `init_pipe` service of USB manager to search and allocate pipe for further data transfers.

2. Driver performs the protocol specific mass storage reset of the device and checks if it was successful.

3. It then proceeds to execute series of SCSI commands which are explained in next section. The driver tests if the device is ready to perform transactions or not. If the device is found to be ready, the INQUIRY command is sent. Response of INQUIRY command contains many relevant information like SCSI command block structure. The flash drive has command block structure code as zero, which suggests, it is a *direct access block device* with SCSI block command-2 [34] command implementation. Driver then reads capacity of the device. At the end of execution, it initializes SCSI device structure `scsi_device_t` which contains following information

   - Block size on the device. For a typical flash device which is 512 bytes.
   - Last addressable logical block number. The block size multiplied by last block's number yields the capacity of the device.
   - Device attribute related meta data e.g. removable, self powered etc.

10.2.1 Implemented SCSI Commands

Following commands are implemented in the mass storage driver. For details please refer to the official documentations ([34], [35]). Here only the relevant sections are discussed.

- **Test ready device:** The TEST READY command tests if the device is in a good position to accept and process requests. If the device is unable to accept the requests then an explicit device start is required by START_STOP command.

- **Inquire device:** INQUIRY command retrieves many important metadata from the device. The information obtained by the INQUIRY command includes, but not limited to

  - **Device type:** Sequential access, direct access, printer device, optical memory etc. are to name few kind of device types. Flash drive has direct access block device type (00h).

- **Device qualifier:** Current status of the hardware device server. Valid values are shown in table 10.5. A properly working flash mass storage device has value 000b.

Table 10.5: Valid device qualifiers

Value	**Description**
000b	A peripheral device having the specified peripheral device type is connected to this logical unit.
001b	A peripheral device having the specified peripheral device type is not connected to this logical unit.
010b	Reserved.
011b	The device server is not capable of supporting a peripheral device on this logical unit.
100b to 111b	Vendor specific.

- **Read capacity of device:** The READ_CAPACITY command reads the capacity of the device. The data associated with the command block returns block size (typical 512 bytes) with last addressable logical block. Multiplication of these two value yields the capacity of the device in bytes.
  $Total\ capacity\ (in\ Bytes) = (Block\ size) \times (Last\ addressable\ block + 1)$

  The addressable block address starts with zero hence extra 1 is required to count. All data read or write transactions are done in quanta of block size. Any other value will result in packet babble error, which indicates that device tends to send more data to the host controller during read.

  The currently implemented command is READ_CAPACITY10 which has 8 bytes block counter limit. If the number of logical blocks exceeds the maximum value that is able to return in 4 bytes, the device server returns FFFFFFFFh. For large storage devices which exceed this limit READ_CAPACITY16 command should be used.

- **Read logical block:** Read logical block reads a given range of blocks from the device. It takes two arguments (apart from other non-relevant arguments), starting block address and number of consecutive blocks to be read. It also has many flavors e.g. READ6, READ10, READ12, READ16 and READ32, each with increasing number of features and complexity. READ6 can only address up to 2 GB and for newer designed device READ10 is recommended for reading purposes. For current implementation we use READ10. READ10 contains many protection related flags and fields. Since a typical flash drive does not support them, most of these fields are set to zero. The READ10 command is executed in 3 phases. In phase one, CBW containing READ10 command is send to device. Expected data bytes are transferred in next stage. In the last stage status is read and checked against to ensure transfer was successful. Reading an invalid block address or range will result in endpoint stall.

- **Write logical block:** Like READ command WRITE also has many flavors which are comparable to the READ commands e.g. WRITE6, WRITE10, WRITE12, WRITE16 and WRITE32. For current implementation we have implemented WRITE10. It takes two arguments (apart from other non-relevant arguments), starting block address and number of consecutive blocks to write. Similar to READ command WRITE is also completed in three phase. During the fist stage, SCSI command is sent to the device on OUT endpoint. In second stage, data is provided by HC to the device on OUT endpoint. In last stage, status of command is read from the IN endpoint. A good status ensures that data is written successfully to the device.

- **Synchronize cache:** Synchronize cache command synchronizes the device cache to the persistent storage. Often to improve read or write performance data is fetched from cache. But for proper storage data is written back to storage in timely manner by synchronize cache command. Applications which have strict consistency requirements should write directly to the device.

All commands and device related information is crosschecked with Linux and found to be same. it ensured the correct implementation of the commands.

10.3 Driver Server's Services

The USB device driver's primary task is to execute storage requests of an application on the device. Hence the implemented SCSI commands are also exported as services offered by the driver server. The following services are implemented in the driver server

1. **Probe:** Probe provides a way for USB manager to check and ask device server if it is in position to accept new device. It takes are required

parameters and return ACCEPT or REJECT for a particular request.

Function call: `probe(uint8_t dev, uint8_t class, uint8_t subclass, uint8_t protocol);`

2. **Disconnect:** Similar to probe, disconnect is called from USB manager to notify driver server that the device has been removed. Upon notification driver then proceeds to perform clean ups.

   Function call: `disconnect(uint8_t dev);`

3. **Get SCSI device:** All the storage operations are done on SCSI device. An application first have to obtain the SCSI device structure from the driver. The SCSI device contains capacity, block size, last addressable block numbers, LUNs etc. which helps in an application to set the requests.
   Function call: `void get_scsi_dev(scsi_device_t dev);`

4. **Read logical block:** Read logical block service reads the data from a given range of consecutive blocks from the device. The I/O buffer is provided by the application. The cache flag indicates to read from cache or from storage. Zero means that device server may read the logical blocks from volatile cache, non-volatile cache, and/or the medium. One signifies that device shall read the logical blocks from the medium. If cache contains a more recent version of a logical block, the device shall write the logical block to the medium before reading it.

   Function call:
   ```
   void read_scsi(scsi_device_t dev,
                  uint32_t start, uint32_t num,
                  uint64_t buff, uin8_t cache);
   ```

5. **Write logical block:** Write logical block service writes the given data on a given range of consecutive blocks on the device. Similar to read, I/O buffer is provided by the application. Cache is same as explained above.

   Function call:
   ```
   void write_scsi(scsi_device_t dev,
                   uint32_t start, uint32_t num,
                   uint64_t buff, uint8_t cache);
   ```

6. **Synchronize cache:** This service explicitly purges the volatile cache of the device so that data is moved to the persistent storage.

   Function call: `void scsi_sync(scsi_device_t dev);`

Chapter 11

Evaluation

This chapter will give an overview of the implemented distributed driver system performance. The current prototype contains a device driver for flash mass storage device. All experiments are done on the AMD 2X2 core box (nos4) which has two dual-core Opteron 2220 cores [36]. It has 8 GB of RAM which is equally shared between the 2 sockets which are in different proximity domains.

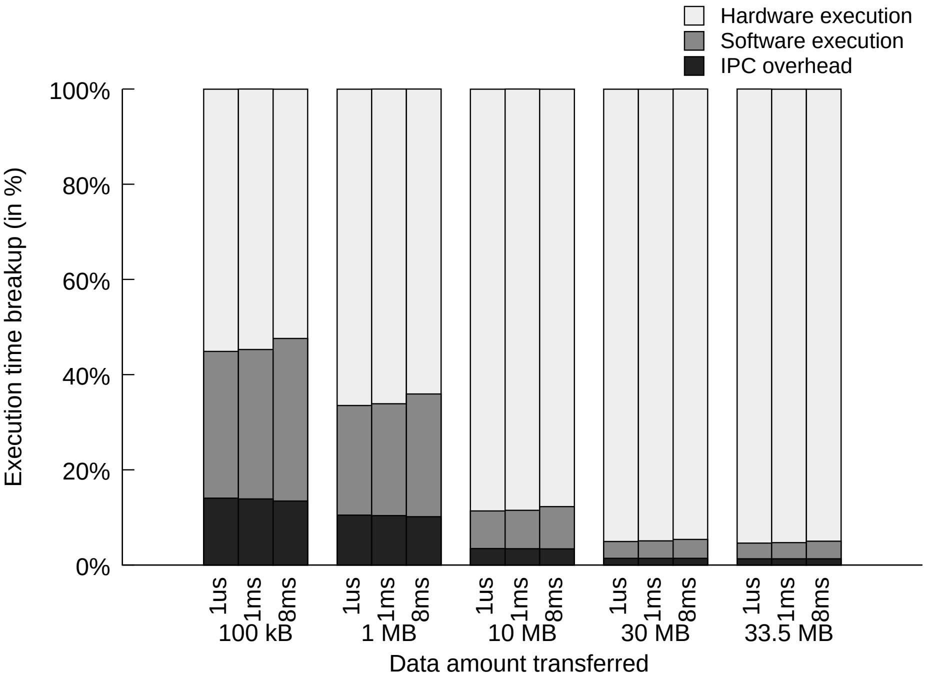

Figure 11.1: Execution time breakup

The figure 11.1 gives an overview of different stages of execution and how they contribute to the overall execution measurements. The graph is

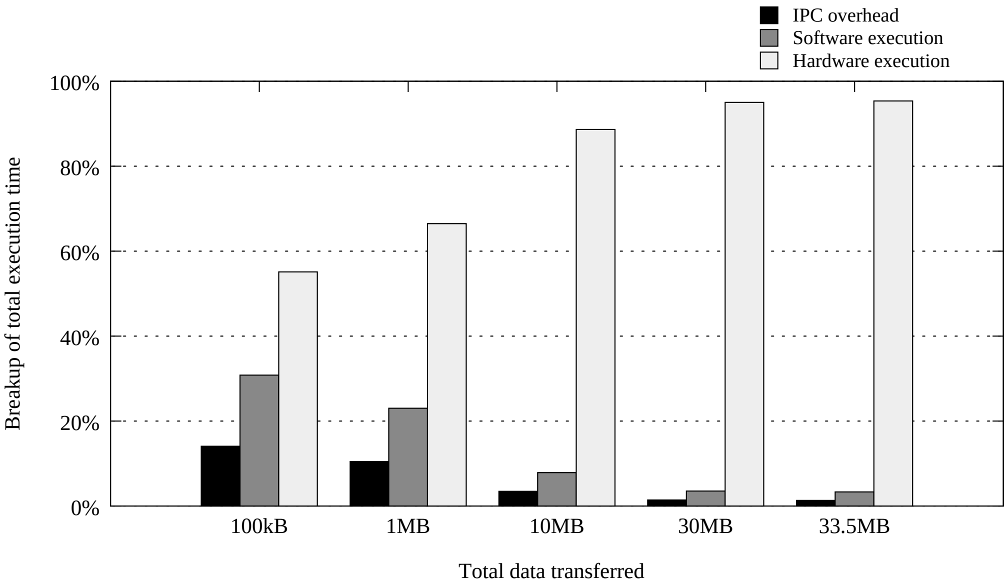

Figure 11.2: Execution time breakup on 1 μsecond

for read performance on the flash device. Value shown here are averaged over 5 different runs. On the x axis it shows the amount of data read from the device and on the y axis it shows three major phases of execution and how they contribute to overall process. These three phases are: execution in software, execution in hardware and IPC done to communicate between the domains. The software execution time represents the time taken to setup queue, buffers and linkage in software. The hardware execution time shows the time from when software enqueues the request into the hardware queue, to when software receives an interrupt on completion (IOC). The hardware time roughly tells the time taken by the EHCI controller to execute the request.

The x axis also shows interrupt frequency data. The EHCI host controller generate interrupts on a specific frame frequency. This parameter is tunable and set before the controller is started. The default frequency used is 8 micro frames, which equals to 1 millisecond. All interrupts except controller hardware errors are generated on the frame boundaries. IOCs are cumulative. For every read test we have taken data on three interrupt frequency setup, 1 micro-frame or 125 μsecond (minimum possible on the EHCI controller), 1 millisecond (default on EHCI controllers) and 8 millisecond (maximum possible on the EHCI controller). As evident from the figure 11.1 that variation in host controller frequency does not make any significant difference. On 125 μsecond frequency setup the hardware generates interrupts at every micro-frame and maximum throughput is observed there.

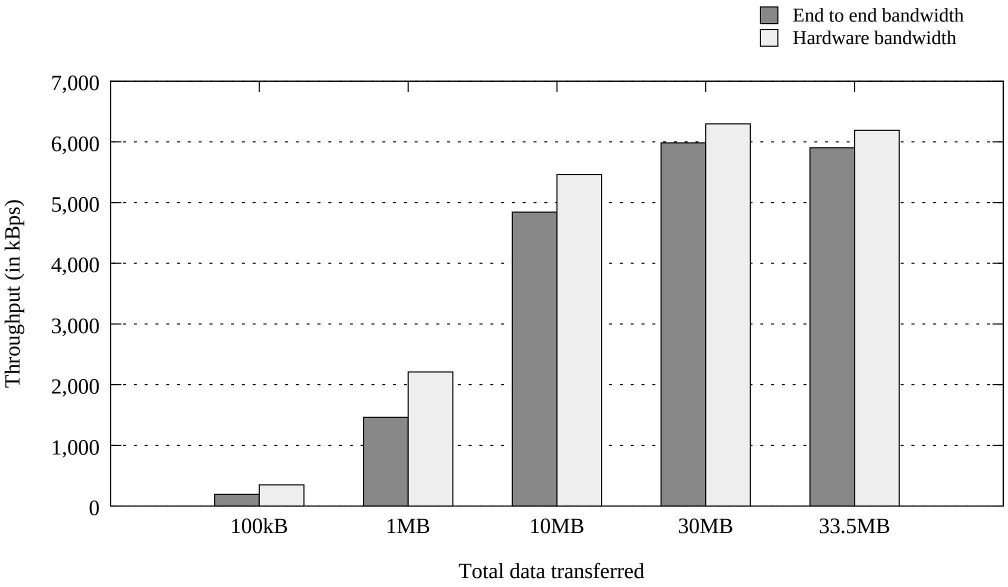

Figure 11.3: Bandwidth observed

The graph also shows that for small read requests the IPC overhead is significant (14%) but as we move on to read bigger chunks it reduces to less than 1%. For bigger read requests hardware execution cost significantly dominates the overall cost. 33.5 MB is the biggest request that driver can issue in one read request with READ10 command. The figure 11.2 gives more detailed overview. For small read requests like 100 kB the IPC contributes to as high as 14.06% to the overall execution cost. For large requests the overhead is amortized and this ratio is less then 4% (3.47%, 1.42%, and 1.31% for 10 MB, 30 MB, and 33.5 MB respectively). On the other hand with increasing request size, the hardware execution time contributes significantly to the overall cost. For 10 MB, 30 MB and 33.5 MB read it contributes as high as 88.63%, 95.03%, and 95.37% respectively.

The figure 11.3 shows the observed bandwidth in the system at the top of the software stack. The figure shows two type of bandwidths. First one is called *end to end* bandwidth. It represents the bandwidth observed by end client after issuing a read request. It includes all three phases of mass storage protocol, command, data, and status check. The hardware bandwidth represents the data processing speed of the hardware. It is the actual time which a request spent in hardware. As evident from the figure the bigger block read request leads to better bandwidth performance but after a while the gain is not significant. For example increasing block size from 100 kB to 1 MB (which is a 10X increase) leads to a big bandwidth improvement of

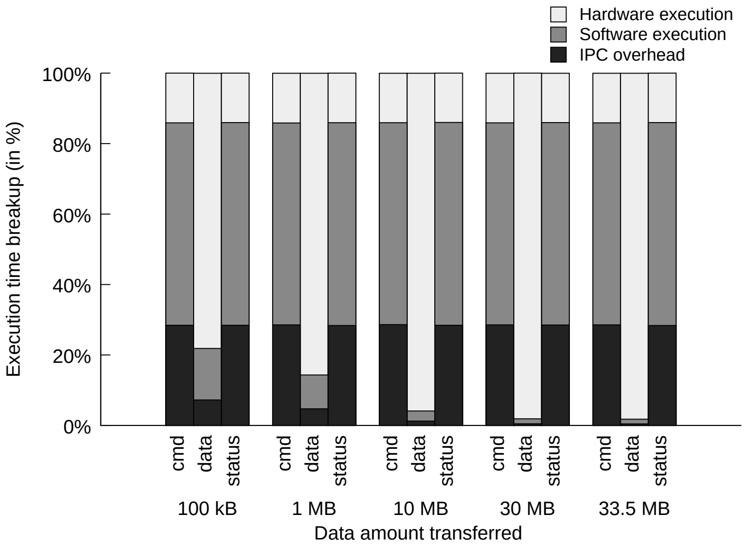

Figure 11.4: Break up of read command execution stages

nearly 7.61X (661%) with 0.76 improvement factor. But increasing block size from 10 MB to 33.5 MB (which is a 3.35X increase) leads to bandwidth improvement of only 1.21X (21%) with 0.36 improvement factor. The increase in size from 30 MB to 33.5 MB actually slightly decreases the bandwidth. The hardware throughput shows a similar pattern with improvements in the bandwidth with larger chunk sizes.

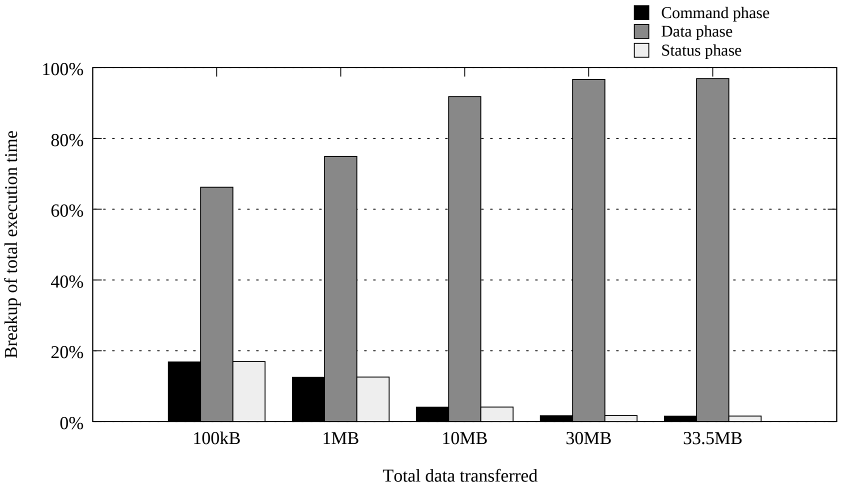

Figure 11.5: Break up of time spent in read command request stages

The Linux reports 13 MBps bandwidth with dd command utility on the

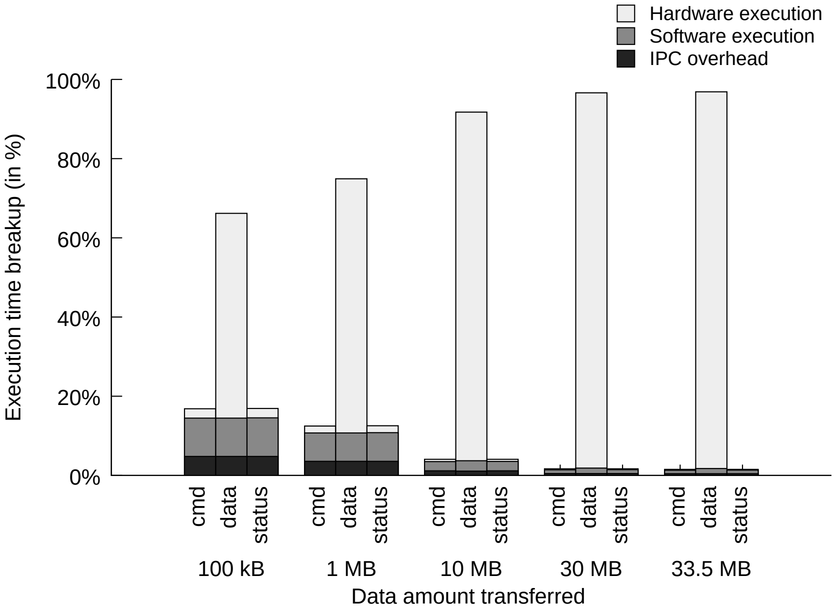

Figure 11.6: Break up of read command execution stages as ratio of time spent in each stage

same hardware. Comparing to Linux we have only achieved a fraction of bandwidth (1/2). As the software's contribution while executing a request is pretty much insignificant compared to hardware contribution, the hardware cost dominates the overall execution time. The peak hardware performance is 6.29 MBps at 30 MB read (it decreases from it to 6.19 MBps for 33.5 MB read). The potential reason for the low hardware performance is still unknown and requires additional efforts in this direction.

The read command is executed in three different stages. In first stage command is sent to the device. In second stage the data is moved across the USB bus and in third stage the command status is fetched from the device. The figure 11.4 shows in detail the amount of time spent in three different stages as with IPC, software and hardware execution phases. As evident from the graph that the IPC, software and hardware execution phases contribute uniformly to command and status stages of read command with approximately 28%, 57%, and 14% time spent in each of them respectively. But the data stage of read command vary depending upon the size of read request. With the larger requests the data-hardware combination becomes the dominating factor with value as high as 98.19% in 33.5 MB read test.

The figure 11.5 shows the time spent in each stage of read request (command, data, and status). As expected data moving stage is dominating in every size of read request. Even for 100 kB read, 66.20% time is spent in data moving stage. The figure 11.6 shows the data presented in above 2 graphs in one place.

In the currently implemented system the typical device enumeration time is 8 milliseconds. It is the total time spent in USB manager from the device plug notification to just before when USB manager tries to connect with device driver server as a client. It includes reading configurations, interface and endpoint descriptors, allocating and assignment of device address, insertion of device node in USB tree etc.

Chapter 12

Conclusions

In this thesis we have presented a distributed driver infrastructure for USB on top of Barrelfish, a multikernel operating system. We have divided the whole USB system into three critical modules. First, the core controller driver which manages the host controller hardware and handles all data transfer logic on the USB bus. Second, the USB manager which handles the device and bus management related activities on USB. The last one is client driver, which handles the device services implemented by the device hardware like storage, mouse, or keyboard etc. We run all these modules in separate domains and requests are processed by explicit message passing. There are several advantages of this breaking such as less interference with each other, more isolation and protection etc. It helps in future revisions of system. For example, for the next version of host controller (xHCI [31]) one can just implement the HC driver and can provide services on same API. The remaining system can remain unchanged. Also this design gives us flexibility to manage the system workload more efficiently by choosing free cores to run these modules and migrate among them, if required. A failed module *can* be restarted transparently. The implemented prototype is still in a development stage and requires more efforts before these goals can be fully achieved.

Chapter 13

Future Works

In this chapter we will present some directions for future works.

1. The end to end bandwidth is still poor as compared to the Linux systems. A proper diagnosis of the overhead is required.

2. The implemented memory management has O(n) complexity which is not efficient. A more efficient memory management architecture is desired.

3. In current implementation the HCD allocates the queue heads and elements in its domain on behalf of client drivers. A notorious driver can potentially crash HCD by requesting a large chunk of read or write request. A single queue head (64 bytes) and element (32 bytes) can handle 20kB worth of data transaction. For larger request only more queue elements are allocated. Hence in more secure implementation, driver should also provide pages on which queue allocation *should* be done.

4. In the current implementation (for the flash mass storage device) most of the configuration options are ignored because often they are the only ones. So there is no choice in terms of assignment. But a comprehensive handling of these options should be done.

5. Linux aggressively reuses queue heads and elements in the queue management. In current implementation for every requests these are allocated and then freed. An efficient re-use policy could lead to improved end to end bandwidth in case of periodic requests.

6. A better SKB integration is required for generating recommendations such as port swapping for overloaded ports. But since USB does not distinguish between internal and external ports a port learning algorithm could be helpful in extracting more useful information about USB system.

7. The current implementation does not support periodic schedule.

8. A periodic monitoring and decision making architecture which could tune USB performance according to workload is missing. It may be responsible for deciding which module starts on which core. In case of overloaded system it can make recommendation to the system to migrate USB modules across the cores.

Bibliography

[1] Intel Terascale computing research program, http://techresearch.intel.com/articles/Tera-Scale/1421.htm.

[2] AMD Fusion Architecture, http://en.wikipedia.org/wiki/AMD_Fusion.

[3] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs, "Embracing diversity in the Barrelfish manycore operating system." in *Proceedings of the Workshop on Managed Many-Core Systems (MMCS)*, Boston, MA, USA, June 2008.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A New OS Architecture for Scalable Multicore Systems," in *SOSP '09: Proceedings of the 22nd ACM symposium on Operating systems principles.* New York, NY, USA: ACM Press, October 2009.

[5] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles.* New York, NY, USA: ACM, 2003, pp. 207–222.

[6] Y. Padioleau, J. L. Lawall, and G. Muller, "Understanding collateral evolution in linux device drivers," in *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006.* New York, NY, USA: ACM, 2006, pp. 59–71.

[7] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Construction of a Highly Dependable Operating System," in *Proc. 6th European Dependable Computing Conference (EDCC-6)*, S. Ceballos, Ed. Coimbra, Portugal: IEEE Computer Society, Oct. 2006, pp. 3–12.

[8] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole Jr., "The exokernel approach to extensibility (panel statement)," in *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI '94)*, Monterey, California, November 1994, p. 198.

[9] J. Liedtke, "On micro-kernel construction," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, pp. 237–250, 1995.

[10] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser, "User-level device drivers: Achieved performance," 2005. [Online]. Available: citeseer.ist.psu.edu/leslie05userlevel.html

[11] K. Fraser, S. H, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe hardware access with the xen virtual machine monitor," in *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[12] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz, "Unmodified device driver reuse and improved system dependability via virtual machines," in *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation.* Berkeley, CA, USA: USENIX Association, 2004, pp. 2–2.

[13] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the xen virtual machine environment," in *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments.* New York, NY, USA: ACM, 2005, pp. 13–23.

[14] A. Menon, S. Schubert, and W. Zwaenepoel, "Twindrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest os drivers," in *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems.* New York, NY, USA: ACM, 2009, pp. 301–312.

[15] V. Chipounov and G. Candea, "Reverse-engineering drivers for safety and portability," in *4th Workshop on Hot Topics in System Dependability (HotDep)*, 2008.

[16] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha, "The design and implementation of microdrivers," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 2, pp. 168–178, 2008.

[17] M. M. Swift, B. N. Bershad, and H. M. Levy, "Recovering device drivers," in *In OSDI*, 2004, pp. 1–16.

[18] M. V. M. B. Ulfar Erlingsson, Martin Abadi and G. C. Necula, "Xfi: Software guards for system address spaces," in *In OSDI*, 2006.

[19] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer, "Safedrive: safe and recoverable extensions

using language-based techniques," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation.* Berkeley, CA, USA: USENIX Association, 2006, pp. 45–60.

[20] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtual architecture: a safe execution environment for commodity operating systems," in *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles.* New York, NY, USA: ACM, 2007, pp. 351–366.

[21] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider, "Device driver safety through a reference validation mechanism," in *In OSDI*, 2008.

[22] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller, "Devil: an idl for hardware programming," in *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation.* Berkeley, CA, USA: USENIX Association, 2000, pp. 2–2.

[23] I. K. Leonid Ryzhyk, Peter Chubb and G. Heiser, "Dingo: Taming device drivers," in *In Proceedings of the 4th EuroSys Conference*, 2009.

[24] S. Dolev and R. Yagel, "Self-stabilizing device drivers," *ACM Trans. Auton. Adapt. Syst.*, vol. 3, no. 4, pp. 1–29, 2008.

[25] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006.* New York, NY, USA: ACM, 2006, pp. 73–85.

[26] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher, "Model checking concurrent linux device drivers," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering.* New York, NY, USA: ACM, 2007, pp. 501–504.

[27] M. F. Spear, T. Roeder, O. Hosdon, G. C. Hunt, and S. Levi, "Solving the starting problem: device drivers as self-describing artifacts." in *Proceedings of the 1rd European Systems Conference (Eurosys)*, Leuven, Belgium, April 2006.

[28] Systems Research Group, ETH Zurich, "Mackerel: A device IDL." 2008.

[29] ——, "Flounder DSL." 2009.

[30] USB 2.0 protocol documentation at, `http://www.usb.org/developers/docs/usb_20_052709.zip`.

[31] Extensible Host Controller Interface (xHCI) Draft Specification for USB 3.0, at http://www.intel.com/technology/usb/xhcispec.htm.

[32] Intel EHCI documentation at http://www.intel.com/technology/usb/ehcispec.htm.

[33] USB forum, Universal Serial Bus Mass Storage Class Specification Overview at www.usb.org/developers/devclass_docs/usb_msc_overview_1.2.pdf.

[34] SCSI forum, SCSI block Commands-2 (SBS-2) at www.t10.org/ftp/t10/drafts/sbc2/sbc2r16.pdf.

[35] SCSI forum, SCSI primary commands-3 (SPC-3) at http://www.t10.org/drafts.htm#spc3.

[36] See AMD Dual-core Opteron 2220 specification at http://support.amd.com/us/Pages/AMDSupportHub.aspx.