The University of New South Wales
School of Computer Science & Engineering

Mungi Device Drivers

Benjamin James Leslie
<benjl@cse.unsw.edu.au>

**Bachelor of Engineering
(Computer Engineering)**

November 2002

Supervisor: Prof. Gernot Heiser

Abstract

Device drivers are an essential component of every operating system, however recent research has shown that they are also the most error-prone. One way of mitigiating the effects of unreliable device drivers is to run them at user-level where they are isolated from the rest of the system. This thesis presents the design and implementation of a user-level device driver framework for the Mungi operating system. A set of user-level device drivers were developed within this framework and shown to have performance comparable to that of monolithic kernels.

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Device drivers are a critical part of every operating system because they enable interaction with the outside world.

This aim of this thesis is to design and implement a device driver framework for the Mungi operating system and to implement a set of device drivers within this framework.

Some of the major points addressed in the design are:

Flexibility: The device driver framework should be flexible enough to cater for the wide variety of existing hardware and new hardware that is yet to be created. The framework must be portable, so that it can run on the different architectures supported by Mungi. The framework must be flexible enough to stop, stop and install new device drivers while the system is running.

Developer Time: Developing any new software takes time, and with the multitude of devices available it is advantageous to reduce the time required to development device drivers. One of the possible approaches explored is re-using existing device drivers. Another approach is to reduce the complexity of drivers to make development easier.

Protection: Mungi is designed to be a secure operating systems. The device driver framework should integrate in Mungi's existing security framework.

Robustness: As with any software, device drivers in Mungi should be stable and bug free. The thesis explores techniques for decreasing the error-rate, and for isolating device drivers, so that in the case of an error, its effects are mitigated.

Performance: Overall system performance depends on the performance of the input–output subsystem. Mungi device drivers must be fast.

Early in the design of a framework, it was decided that, due to both the structure of Mungi, and the nature of device drivers, it would best to run device drivers at user-level. The motivation behind, and consequences of, this decision is explored in the following chapters.

A framework to support user-level device drivers was designed and a prototype developed. A set of some device drivers was implemented using this framework. The performance of these drivers was measured and compared to the performance of device drivers in the Linux operating system.

1.1 Overview

Chapter 2 describe the function of device drivers within the operating system, how they interact with the hardware and how they interact with other parts of the system. Chapter 3 explains the motivation for running device drivers outside the kernel and the consequences this has on a device driver framework. Chapter 4 examines how device drivers have been implemented in other operating systems.

Chapter 5 introduces the Mungi operating system, describing its design and implementation. This chapter also describes the Mungi component system, and some of the improvements made to it during the course of the thesis. Finally this chapter examines previous work on device drivers in Mungi.

Chapter 6 examines the possibility of reusing device drivers, including a review of how other projects have approached this problem.

Chapter 7 describes the implementation of a domain-specific-language which was used to improve the development of device drivers.

Chapter 8 explores the different modes of operations possible for transferring data between device drivers and other components in the system.

Chapter 9 presents a design for a Mungi device driver framework. The chapter concentrates on the mechanisms that the Mungi kernel needs to provide.

Chapter 10 describes the implementation of the Mungi kernel changes required to support the device driver model.

Chapter 11 introduces the different I/O busses used in current workstations, provides a design of an interface to them, and describes the implementation of a driver which implements the interface.

Chapters 12, 13, and 14 cover the design of a stream, packet and block interface respectively. Each of these chapters also presents the implementation of a device driver using the designed interface.

Chapter 15 presents the results of performance testing.

Chapter 16 discusses how the performance results could be improved, some of the limitations of the current implementation, and some of the difficulties encountered during the course of the thesis.

Finally, Chapter 17 contains some concluding remarks, and some ideas about future work.

Chapter 2

What is a Device Driver?

At the most fundamental level a *device driver* is simply a piece of code which controls a hardware device. The driver provides an interface so that programs can access the functionality of a device without knowing the details of the hardware.

2.1 Device registers

Most of the devices' functionality is controlled by manipulating its registers. Some of these registers are simply used to store data, in the same way CPU registers are used. However, unlike CPU registers, accessing some registers can cause the device to perform some operation.

Figure 2.1 shows the register layout for a hypothetical disk controller[1] device. In this example, registers one and two are simply used to store data, specifically the parameters to a disk operation. The actual disk operation is performed when the device driver writes the operation number into register three. The status of the controller can then be checked by reading register four.

0x1	Sector
0x2	Count
0x3	Operation
0x4	Status
0x5	Data

Figure 2.1: Example disk controller registers.

So that a device driver can access the registers, the system must export a programming interface. System designers solve this problem using two main techniques.

[1]This example is based on the IDE controller device, however it has been simplified for clarity.

I/O Ports

One approach, which is used in the Intel IA-32 platform [Int02], is to map the device registers into an I/O address space which is distinct from the physical address space. This is usually known as the I/O port space. The programmer then uses the `IN` and `OUT` instructions to access the registers.

Continuing the disk controller example, Figure 2.2 shows how a device driver would access it using the I/O space abstraction, assuming it was mapped into I/O space at address 0x100.

```
out(0x101, 23);     /* Start at sector 23 */
out(0x102, 5);      /* Transfer 5 sectors */
out(0x103, 1);      /* Initiate a read    */
status = in(0x104); /* Read the status    */
```

Figure 2.2: Example use of `in` and `out` instructions.

Memory mapped registers

An alternative model for accessing device registers is to map device registers into the normal physical address space (Figure 2.3). The programmer is then able to use normal memory instructions to access and manipulate device registers. This approach is used by MIPS-based and Alpha-based platforms. The IA-32 platform also uses this method in addition to its I/O address space.

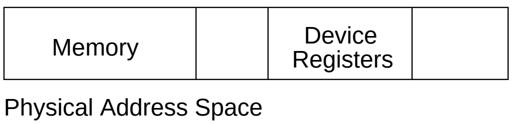

Figure 2.3: Example disk controller registers.

From the device driver programmer's point of view this is not really a big change. Example code for accessing our imaginary disk controller's registers, if it were mapped into physical memory at address 0x4000, is displayed in Figure 2.4.

```
*(char*) 0x4001 = 23;      /* Start at sector 23 */
*(char*) 0x4002 = 5;       /* Transfer 5 sectors */
*(char*) 0x4003 = 1;       /* Initiate a read */
status = *(char*) 0x4004;  /* Read the status */
```

Figure 2.4: Example use of memory mapped registers.

Memory mapped device registers must be treated differently from regular memory because of the possible side effects, such as initiating a data transfer, that can occur from reading and writing to a device register. Caching accesses to device registers is clearly a problem. If we wish to initiate a disk transfer it is no good if the write is stored in the cache. It is equally problematic if, when reading the status register, a cached copy is returned to the device driver, rather than the current status.

Architectures provide different mechanisms for solving this problem. The IA-32 provides provides a set *memory type range registers*, which allow the operating system to specify the caching attributes for different ranges of the physical address space. In Alpha platforms using the 21174 [DEC97] chip-set, only the lower half of the physical address space is accessed through the cache, leaving the upper half free for memory mapped devices. The MIPS architecture handles caching on a per-page basis by setting a caching attribute in the TLB entry.

Another related problem is that some architectures, such as the Alpha, are only able to reference memory with aligned 32-bit and 64-bit references; there is no instruction to read a single byte from memory [DEC97]. Unfortunately many devices have 8-bit or 16-bit registers and reading a whole word would cause unintended side effects. To avoid this problem, the Alpha uses a technique called *sparse space* (see Section 11.5.2) to access device registers. In this technique some of the address bits are used to encode the size of data transfer.[2]

2.2 Interrupts

Device are typically much slower than the CPU, consequently, device drivers are often left waiting for the device. For example after writing the `read` command to the disk controller, the driver needs to wait until the controller is ready to transfer the data. The simple way to do this is to *poll* the device until it is ready. Our example disk controller sets its status register to 1 when it is ready to transfer data. Figure 2.5 shows an example of the device driver polling the status register.

```
while ( *(char*) 0x4004 != 1 )  /* Check the status register */
  {
  }
/* Do data transfer */
```

Figure 2.5: Example of polling.

Polling a device is undesirable because many CPU cycles, which could be used for other processing, are wasted while spinning on a status register. The alternative to this approach is to use *interrupts*. Any device capable of generating an interrupt is attached to an interrupt request (IRQ) line. When a device generates an interrupt it asserts a signal on the IRQ line. Raising this signal will then, generally, cause an interrupt on the CPU. An interrupt causes the CPU to execute an interrupt handler routine.

Luckily our fictional disk controller is able to generate an interrupt when it ready to transfer data. The allows our device driver to be rewritten as shown in Figure 2.6

There is not always a one-to-one mapping between IRQ lines and CPU interrupts. The Alpha architecture has up to 61 IRQ lines which are routed through up to four different CPU interrupts [DEC97].

When a CPU interrupt is raised, the programmer must determine which IRQ line has been raised, by interrogating the IRQ logic. The IRQ logic usually allows individual IRQ lines

[2]Some Alphas have *byte word extensions* (BWX), which allows the processor to address individual bytes, avoiding the need for sparse space.

```
interrupt_handler()
  {
     /* The device must be ready if we recieved an interrupt */
     assert ( *(char*) 0x4004 == 1 )

     /* Do data transfer */
  }
```

Figure 2.6: Example interrupt handler.

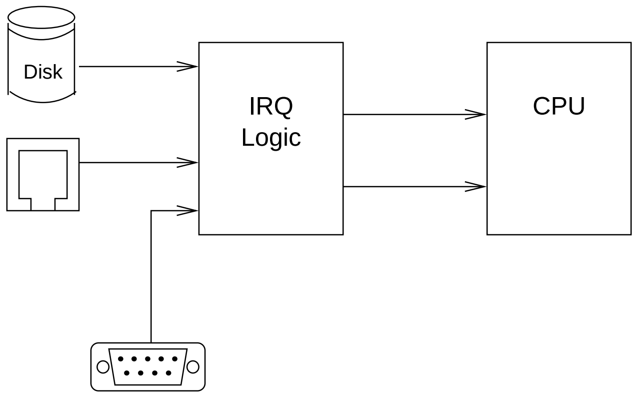

Figure 2.7: Simplified IRQ logic.

to be masked. A masked interrupt will not generate a CPU interrupt, even if the IRQ line is asserted.

When an interrupt occurs, the code handling it must acknowledge that it has received the interrupt. This generally requires acknowledging both the IRQ controller, to prevent it generating more CPU interrupts, and the actual device so it no longer asserts the IRQ line.

Although some architectures provide many IRQ lines, some support as few as 15. This is limiting as there is often more than 15 devices in a system that need to generate interrupts. To get around this problem multiple devices must share the same IRQ line.

2.3 Direct memory access

The final aspect of hardware devices is how data is transferred between the device and physical memory. One method of doing this is to simply have the device driver copy data between device registers and physical memory. This is called *programmed I/O* (PIO). When our hypothetical disk controller is ready, one byte can be transferred every time the data register, register 5, is read.

```
/* Do data transfer */
for ( i = 0; i < TRANSFER_SIZE; i++ )
  {
     buffer[i] = *(char*) 0x4005;
  }
```

Figure 2.8: Example programmed I/O.

As with polling, PIO consumes a lot of CPU cycles, which could be put to better use. *Direct memory access* (DMA) is mechanism that supports the direct transfer of data between the physical memory and devices in a system, without the intervention of the CPU. This has the obvious, and substantial, benefit of allowing the CPU to continue executing during data transfer.

The exact method for initiating a DMA transfer is dependent on the bus architecture, and platform, being used.

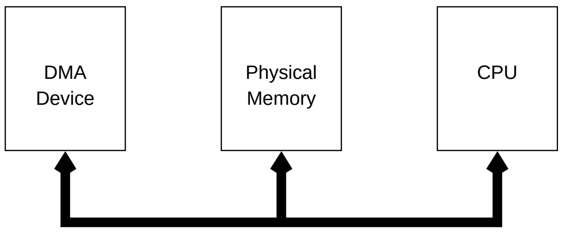

Figure 2.9: Direct memory access.

2.4 Mechanisms for data transfer

Device drivers do not only interact with hardware, they also have to interface with programs. Data transfer usually achieved by the device driver storing data in a private buffer, and copying it into the programs buffer when required.

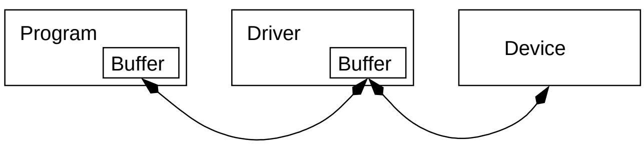

Figure 2.10: Data buffered in the device driver.

This interface suffers the same inefficiencies as PIO: the CPU cycles are devoted to simply moving bytes from one part of memory to another. The alternative is that the device driver doesn't buffer the data, but instead transfers the data directly from the device into the programs data buffer. This process is known as *zero-copy*, and is shown in the Figure 2.11.

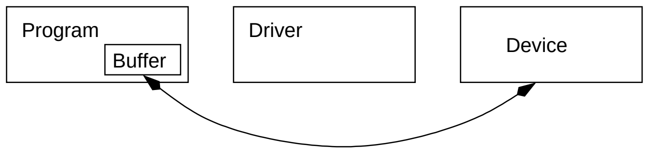

Figure 2.11: Zero-copy operation.

2.5 Data transfer abstractions

Device drivers provide abstractions for the of data between programs and hardware devices. Despite the proliferation of different devices most fall into one of the following categories:

Stream based devices simply transfer a set of bytes, with no fixed structure placed on the byte stream. Devices such as serial ports, printers, keyboards and mice fall into this category.

Packet based devices transfer data in discrete, variable sized, packets. Incoming packets are often multiplexed to different programs. Network cards are generally packet based.

Block based devices transfer data as fixed sized chunks. Storage devices such as hard drives and CD-ROMs are typical examples of block based devices.

2.6 Device control

The transfer of data between between driver and client is only one aspect of interaction; devices also require various *control* methods. A typical example of a control method would be changing the baud rate on a serial device. While many devices share common methods of transferring data, there is a wide variety of different control commands which must be taken into account.

Chapter 3

User-level Device Drivers

Device drivers are usually run as part of the kernel in most operating systems, which means that device drivers have full access to the system, including kernel data structures. An alternative is to run device drivers at user-level.

3.1 Motivation

3.1.1 Isolation

The major advantage to running devices at user-level is the drivers can be isolated from both the kernel, and each other. Most drivers are very complex pieces of code, and as a result are prone to errors. A study conducted at Stanford University found that in the Linux kernel the majority of bugs were found in the device drivers, rather than the core kernel [CYC+01]. Figure 3.1 shows that device drivers have error rates up to 7 times higher than the rest of the kernel.

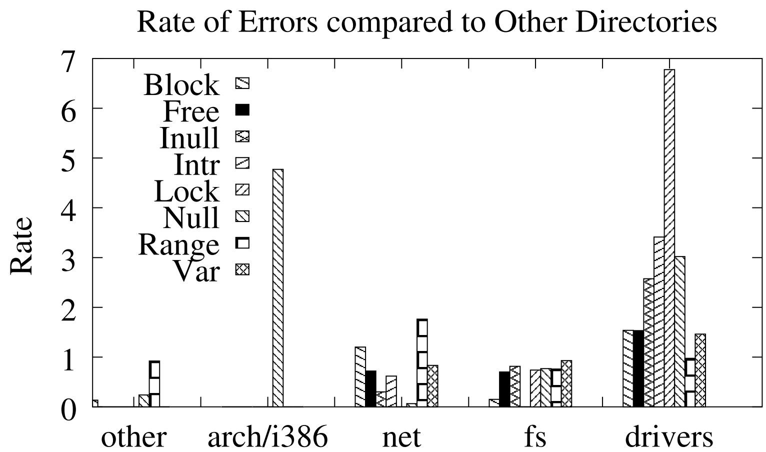

Figure 3.1: Relative error rates in the Linux kernel. Reproduced from: [CYC+01]

One of the reasons for this high error rate is that device drivers are often written by third parties, who do not have a thorough understanding of kernel internals. The other major cause is that both hardware devices and kernel internals are generally very complex, which increases the difficulty of making bug free device drivers [SMLE02].

Moving drivers out of the kernel improves this in two ways. Firstly it forces the kernel to provide clean, documented interfaces for device drivers to use, which will reduce the complexity involved in writing device drivers. More importantly, if a device driver does have a bug, it is not able to interfere with the kernel.

3.1.2 Flexibility

User-level programs are given a lot of flexibility: they can be easily stopped and restarted, new programs can be easily installed and when a program crashes a debugger can be used to trace down the cause. These features are generally not available to kernel level code. These features all become available when device drivers are run at user-level.

Modern hardware allows various devices, such as PC Card and *universal serial bus* devices, to be connected to the system at run-time. If device drivers are simply user-level programs loading device drivers dynamically becomes as easy as starting a new program.

3.2 Accessing device registers from user-level

If a device driver is running inside the kernel no special action is needed to access the device registers. This is no longer the case if a device driver runs at user-level. To allow otherwise unprivileged code to access device registers, platforms must provide a protection mechanism. When device registers are memory-mapped, protection can be handled by the *memory management unit* (MMU); only the kernel can directly access physical memory, other processes can access physical address space when it is mapped into their virtual address space.

To continue the disk controller example presented in the previous chapter, we now wish to write a user-level device driver. The driver needs to map the register into its virtual address space. Having done this driver can then simply access registers as before, albeit at a different memory offset.

```
/* Map the device registers into our virtual address space
   at 0x10000 */
map_phys_to_virt(0x4000, 0x10000);
* (char*) 0x10001 = 23;     /* Start at sector 23 */
* (char*) 0x10002 = 5;      /* Transfer 5 sectors */
* (char*) 0x10003 = 1;      /* Initiate a read */
status = *(char*) 0x10004;  /* Read the status */
```

Figure 3.2: Example use of user-level memory mapped registers..

Protection granularity is limited to the smallest supported page size. This is problematic if multiple devices' registers are mapped into the same page of physical memory. For example, a serial drivers' registers could be into the physical address space at 0x4050, which is on the same page as our disk controllers' registers. As a result, even though our disk controller driver and serial driver are in separate user-level programs, they can still interfere with each other by directly accessing the other devices' registers.

3.3 Exporting interrupts to user-level

A user-level device driver needs to have some way to receive interrupts from the hardware. The kernel needs to provide a way to allow device drivers to register interest in an interrupt, and to provide a mechanism to deliver this interrupt to a user-level device driver. It is also necessary to provide some mechanism, to allow a device driver to acknowledge and mask interrupts.

Apart from these simple functional requirements there is an important performance requirement: interrupts must be delivered quickly. To illustrate this fact, consider a simple serial port device. These devices generally have a 16 byte *first-in first-out* (FIFO) buffer, which is filled as data is read from the serial line. When then data in the FIFO reaches a trigger point an interrupt is generated.

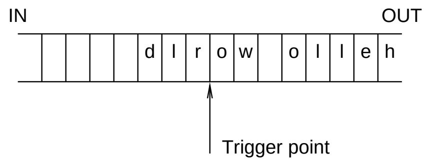

Figure 3.3: A sixteen byte FIFO, with the trigger point set to 8-bytes.

Once the interrupt has been triggered, the device driver must read the data from the FIFO before it fills up. If the interrupt is not serviced quickly enough then overflow will occur, and data will be lost (Figure 3.4). This means that user-level device drivers must be able to service interrupts quickly.

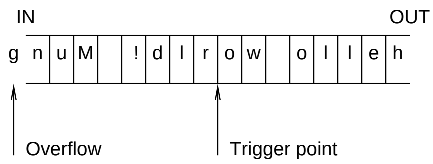

Figure 3.4: Data loss due to FIFO overflow.

3.4 User-level DMA

Setting up devices to perform DMA transfers from user-level is difficult because the device directly accesses physical memory, whereas a user-level device driver works with purely virtual memory.

The device driver needs to be able to perform virtual-to-physical memory lookups. The operating system's virtual memory system needs to provide some interface to allow user-level device drivers to perform this translation.

Simply obtaining the physical address of some memory is not sufficient to enable DMA. The device driver also needs some way of ensuring that the virtual memory is fixed in physical memory for the duration of the DMA transfer.

3.5 Secure DMA & I/O MMUs

A problem with allowing user-level devices to perform DMA is security: DMA interaction provides an opportunity for device drivers to bypass the normal memory protection found in a system.

For example, when setting up a DMA transfer with our disk controller, instead of using in the physical address of our pinned data buffer, we could, accidently, use the address of the kernel's heap. This obviously weakens the major motivation for user-level device drivers: isolation.

This is a drawback of the available hardware mechanisms, and is not something that can be fixed in software. Fortunately not all architectures map DMA addresses directly to physical memory. Instead they use an I/O MMU to virtualise the DMA addresses. This is shown in Figure 3.5. The I/O MMU is completely separate from the normal MMU, and the mappings it provides are totally unrelated.

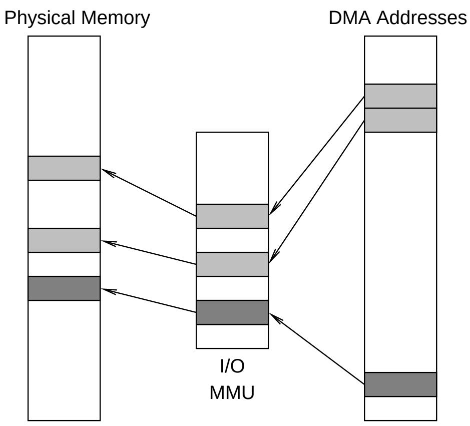

Figure 3.5: DMA addresses translated through an I/O MMU.

The I/O MMU works in a similar way to a normal MMU. The translation are stored in a page table in the host's main memory. When a device performs DMA the translation is looked up in the host's page table. If there is no valid translation available a fault is raised and the DMA is cancelled. An I/O *translation look-aside buffer* (TLB) caches valid translation so a page table lookup is not required for every DMA.

The use of an I/O MMU allows the kernel to isolate itself from devices performing DMA. Unfortunately the I/O MMU is shared between all devices in the system, so it is not possible to isolate device drivers from each other.

3.6 Inter-process communication

Device drivers are of no use unless they can be accessed by other programs in the system. When drivers reside within the kernel user program's can access the device driver with system call, as shown in Figure 3.6.

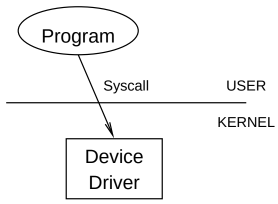

Figure 3.6: User access of a kernel device driver.

If the device driver is also at user-level some form of *inter-process communication* is required to allow programs to call the device driver (Figure 3.7). This presents another potential performance problem. If communication between different user-level programs is slow then I/O performance it going to suffer. Any user–user communication must pass through the kernel, so this interface is always going to be slower than a system call.

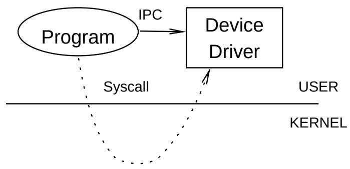

Figure 3.7: User access of a user-level device driver.

3.7 Data transfer at user-level

Transferring data between a user-level program a device driver running in the kernel is fairly easy. The device driver can access the page tables and can easily convert a user supplied virtual address into a physical address. This allows the device driver to read or write data directly from a user's buffer.

A device driver running at user-level obviously needs some other mechanism for transferring data to and from another program. One method for doing this could be passing all data across the IPC mechanism, which is not ideal because it will involve unnecessary copying of data.

The other option is to share the data buffer between the device driver and the other program. This avoids the need for data copying, although it does introduce the need for some method of synchronisation between the driver and user program.

Chapter 4

Related Work

4.1 Linux

Linux is examined as a representative example of a free, UNIX-like, operating system.

4.1.1 Device registers

Linux device drivers, as with most UNIX-like operating systems, execute inside the kernel. Accessing device registers from kernel mode is reasonably straight forward, a device driver can simply directly access the memory or I/O ports required. The problem with this is portability, as illustrated by the Alpha's use of sparse space. To ensure drivers are portable, Linux provides a set of macros to access I/O ports and a separate set of function to access memory mapped registers [Rub98]. Linux allows device drivers to reserve a set of registers (either a range of memory, or a set of I/O ports), however this does not offer any real protection from badly written device drivers.

4.1.2 Interrupts

In Linux, a device driver registers for an interrupt by calling `request_irq`, indicating the IRQ it wishes to register for and a function that processes the interrupt. Since the device driver is running inside the kernel, when an interrupt is received by the kernel it simply calls the previously registered function. Linux allows an interrupt handler to be register as either fast or slow handlers. Fast handlers execute with interrupts disabled, whereas slow handlers execute with interrupts enabled.

4.1.3 Client interfaces

Perhaps the most pervasive interface for transferring data to and from devices comes from UNIX. The Linux operating system also uses these semantics. In UNIX systems each device is represented by a special file and data is transferred using simple, synchronous `read` and `write` calls [TR74]. Support for random access devices is provided by the `lseek` call.

Device specific control is handled through the `ioctl` system call. `Ioctl` is a generic system call used to control all devices in the system, as a result `ioctl` does not have well defined arguments.

4.1.4 Kernel modules

Linux kernel modules allow the kernel to be extended at run-time. This allows device drivers to be dynamically loaded, rather than being compiled into the kernel. This provides the Linux kernel with some of the flexibility usually afforded to user-level programs. Linux kernel modules do not provide any form of protection between modules and the rest of the kernel.

4.1.5 Nooks architecture

The Nooks architecture [SMLE02] attempts to increase stability in the Linux kernel by executing device drivers in a fault-isolating environment. Drivers are isolated by running them inside a *nook*. Each nook is a separate protection domain, however it shares the kernel's address space which allows pointers to be easily shared. It is interesting to see the concepts of a SASOS being applied to the Linux kernel. The nooks architecture allows multiple drivers to execute within the same nook; tightly coupled modules, such as the network card driver and TCP/IP stack, execute within the same nook. This is shown in Figure 4.1.

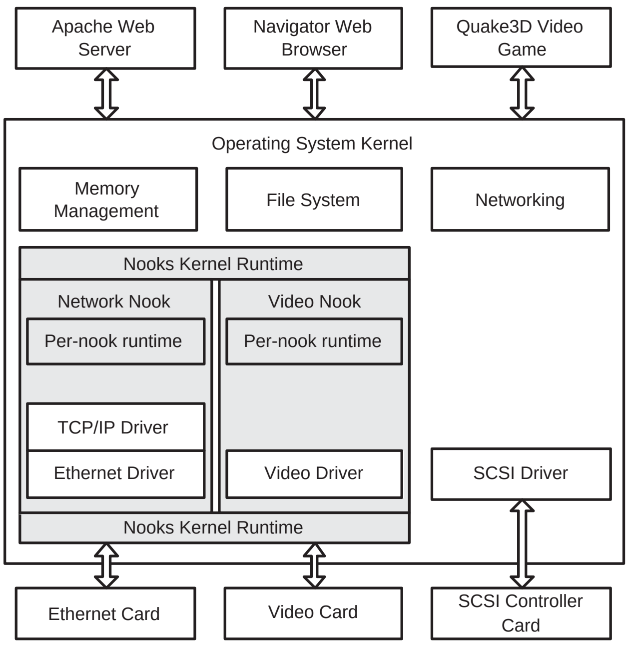

Figure 4.1: Nooks architecture diagram. Reproduced from: [SMLE02]

The nooks architecture checks parameters being passed between the kernel and device driving by wrapping function calls. One of the problems encountered in the implementation of nooks was that “Linux allows drivers to modify main kernel data structures”, which is also one of the problems experienced in reusing Linux drivers in other operating systems.

Unfortunately, the nooks architecture does not address the problems involved with DMA. Another drawback is that the current prototype does not actually execute drivers in separate protection domains, although this cost was emulated in performance testing.

Despite these drawbacks, the benchmarks shown do provide some interesting and relevant data. Although the overhead introduced by the added protection doubled the interrupt handling time, this had only minimal effect on network latency and actually increased network throughput. This increase in throughput is an interesting side effect of increased interrupt latencies. When a network card is running at high speed, the device needs constant servicing from the device driver. In these cases it is much better to simply run with interrupts disabled and regularly poll the device; eliminating the overhead caused by the interrupt handler. Having a slower interrupt latency times reduces the number of interrupts taken, and hence the overall interrupt overhead, resulting in an increase in throughput.

4.1.6 User-level drivers

Although most device drivers execute inside the kernel, Linux does provide some support for accessing device registers from user programs.

There are two methods for accessing registers from user mode in Linux. I/O port access is obtained by calling `ioperm` or `iopl`. Memory is accessed by `mmap`ping */dev/mem*. Calls to `ioperm` and `iopl` are only allowed by the super user, while access to */dev/mem* is controlled by normal file permissions.

Unfortunately this interface does not allow much scope for writing device drivers in user mode. Performing DMA is not easy because although there is a mechanism for pinning memory (`mlock`), there is no way to perform the required virtual-to-physical address translation. Previous research into writing a network-card device driver in user mode extended the kernel to provide this facility [Cha01].

The Linux kernel does not provide any mechanism for exporting interrupts to user-level programs. There have been some attempts to export interrupt to user-level programs via a signal, although no serious device drivers have employed this mechanism [Les02].

4.2 L4

L4 [Lie95] requires all device drivers to be written in user mode. A device driver gains access to device registers by mapping the physical page into its virtual address space. A special protocol is used to create uncached memory mappings [AH98]. Access to I/O ports is handled in a similar way; the I/O port is mapped to a specific task [L4K01]. In this case the granularity is a single port, compared to an entire page of memory.

L4 maps interrupts onto the *inter process communication* (IPC) mechanism. An interrupt handler registers for an interrupt by sending a specially formatted message to the kernel. The interrupt handler thread then waits on a special threadid, which represents a given interrupt. When an interrupt occurs a message is delivered to the thread. This thread is then responsible for acknowledging the interrupt and performing any processing required.

L4 uses a *first-come, first-served* resource allocation policy for both device mappings and interrupts.

4.3 Nemesis

Nemesis is vertically structured, single-address-space operating system. The major aim of the Nemesis operating system is to provide *quality of service* (QoS) guarantees to applications. The main difficultly in providing QoS is accounting for resource usage. Nemesis achieves this by executing the majority of code need by an application in the one domain. This is in contrast to microkernel based system where services are provided by servers executing in different protection domains.

The *Nemesis trusted supervisor code* (NTSC) is very small kernel, providing a scheduler and an IPC mechanism. The rest of the system's functionality is implemented at user-level.

Despite running the majority of code at user-level the NTSC allows device drivers to register an *interrupt-stub* which is run by the kernel when an interrupt occurs. These typically just send an event to the device driver which runs at user-level, however when a device needs interrupts to be serviced quickly this code can do so.

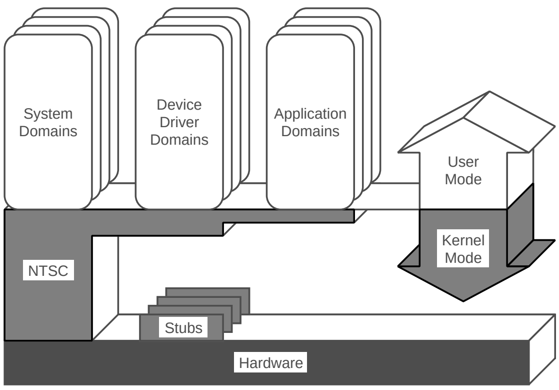

Figure 4.2: Nemesis architecture diagram. Reproduced from: [Bar96]

Nemesis uses a system called *Rbufs* [Bla94] to enable the bulk transfer of data between to domains. In this model when two domains wish to transfer data a pair of control areas is established (see Figure4.3). The control area is a simple circular buffer, which is mapped read-only in one domain, and read-write in the other. To transfer data between domains an `iorec` is placed in the control area. The `iorec` contains a list of *(base, length)* pairs which point to regions of the data area (see Figure 4.4. The other domain is notified when data available through a pair of event channels.

There are two distinct ways in which data can be transferred between *Rbufs*. In *receive master mode*, the domain wishing to receive data places `iorecs` in the buffer and the producer fills them as data becomes available. The other, complementary, mode is *transmit master mode*, where the producer pushes `iorecs` to the consumer, rather than waiting for receive `iorecs` from the consumer.

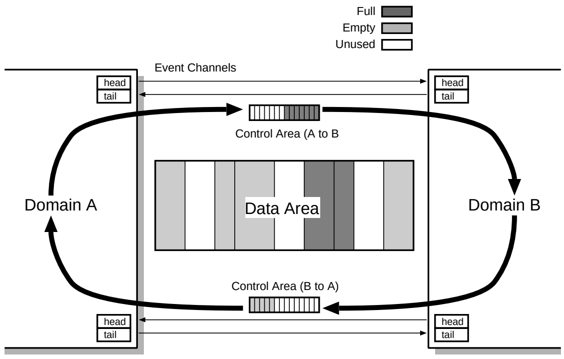

Figure 4.3: *Rbufs* data transfer mechanism. Reproduced from: [Bar96]

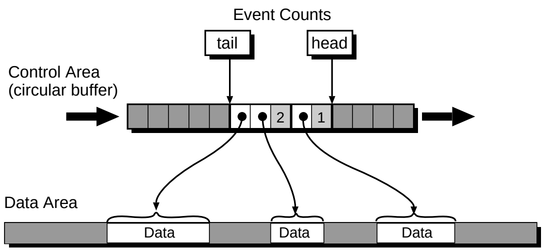

Figure 4.4: The structure of a Nemesis `iorec`. Reproduced from: [Bar96]

4.4 Other operating systems

This section examines some of the novel ways other operating systems have approached some of the problems involved with device drivers.

4.4.1 Plan 9

The Plan 9 system abandons the `ioctl` call in favour of a special *ctl* file [PPTT91]. To perform a control operation, ordinary ASCII strings describing the operation are written to the device's *ctl* file. The major advantage of this is that users, and programmers, do not need to use a special program or system call to control the device. Unfortunately it is just as easy to have an unspecified ASCII interface, as it is to have an unspecified system call interface.

4.4.2 Solaris

Interrupt handling

In the Solaris operating system when an interrupt occurs a thread is spawned to execute the interrupt handler [EKB+92]. Instead of performing a full thread creation for each interrupt, a partially initialised thread is used. This is to minimise the overhead involved in servicing an interrupt.

Most of the thread initialisation work is not performed when the interrupt thread is used. Instead this is delayed until the thread is forced to block, due to lock contention. This reduces the overhead in the common case of no contention.

Zero copy

Solaris implements a zero-copy network stack which uses a technique of remapping pages to transfer data between the user and kernel [Chu96]. A disadvantage of using this technique is that the user must ensure that any buffers are page aligned. Chu saw an increase in performance over the single copy implementation of TCP, however it was greatly affected by the overhead of page remapping on the given system.

4.4.3 Fbufs

Fbufs provide an alternative approach to providing zero-copy mechanisms [DP93]. This technique also uses page remapping to move data between domains. Unlike the mechanism presented by Chu, this method requires the allocation of fbufs which are specifically used for the transfer of data. By reusing fbufs to transfer data, the page remapping can be optimised. The author states that "it is reasonable to require the use of a special buffer allocator for I/O data". I do not agree with this assertion. A program can not always know in advance what data is going to be used for I/O, and the overhead of a copy is now just moved from user data to user fbufs, rather than solving the problem.

4.4.4 Multics

Multics was an early timesharing operating system [CV65]. In the Multics I/O system it is possible to change the driver to use either asynchronous, or synchronous operation using the `worksync` call [FO71]. The only way to check for the completion of an I/O operation is to use the `iowait` system call. This call blocks until the completion of the specified I/O operation. The advantage of this is that it allowed programmers to multiplex I/O calls.

Chapter 5

Mungi

Mungi is a 64-bit research operating system being developed by the Distributed System (DiSy) Research Group at the University of New South Wales. More information on the group's work can be found at: http://www.cse.unsw.edu.au/~disy/. This section provides an overview of Mungi's core abstractions and an introduction to the current L4-based implementation. An overview of Mungi's other features including persistence, distribution, resource accounting, component system and naming service is also provided.

The main feature of the Mungi operating system is that all data, including the kernel's, resides within a single address space. This is in strong contrast to conventional operating systems where each process operates inside its own private address space.

This approach is only practical if the address space is large enough to accommodate all data; 64-bit processors, including the Itanium [Int00a], Alpha [Com98] and MIPS [Hei91] processors provide this feature.

5.1 Mungi abstractions

Objects are Mungi's abstraction for allocation of address space. Objects also form the basic unit of protection in the Mungi system. Each object in the Mungi system is a set of contiguous pages, uniquely identified by its virtual address. The ability to access an object is conferred by a *capability* [VRH93]. In order to be recognised by the kernel, capabilities are often grouped into a *capability lists* (Clists).

In traditional systems, address spaces are used to provide memory protection between different processes; this is not possible in Mungi since all threads share the same address space. Mungi introduces the concept of a *protection domain* which is used to enforce memory protection between different sets of threads. A protection domain is defined by a set of capabilities. The protection domain in which a thread executes is referred to as its *active protection domain* (APD).

To allow threads to cross protection domains in a controlled manner, Mungi provides a protected procedure call mechanism called *protection domain extension* [VERH96] (PDX). Each PDX procedure has a registered entry point and an associated Clist. A thread can execute a call to a PDX procedure with the *PdxCall* system call. The thread's active

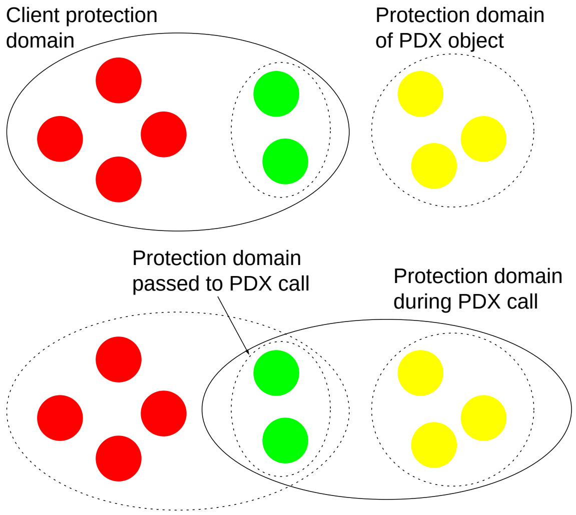

Figure 5.1: The PDX operation. Reproduced from: [Mun02]

protection domain is then augmented with the registered Clist for the duration the call. This operation is illustrated in Figure 5.1. On return from the function thread returns to its original protection domain.

Semaphores provide the basic abstraction for synchronisation in Mungi. The *SemCreate*, *SemDelete*, *SemSignal* and *SemWait* are provided for using semaphores.

Mungi provides *exceptions* to notify a thread when an illegal operation is executed. Threads are able to register an exception handler using the *ExcptReg* system call.

5.2 Mungi implementation

Mungi is currently implemented on top of the L4 microkernel. Current ports run on L4/Alpha [Pot99], L4/MIPS [EHL97], and Itanium. This section gives a brief overview of Mungi's implementation so that the reader is better able to appreciate the design and implementation issues discussed later. This section assumes the reader is familiar with L4 [AH98, Lie95].

Mungi is implemented with two L4 tasks, a *RAM pager* (Rpager) task and the Mungi kernel task (see Figure 5.2. Rpager is responsible for providing memory mappings for the rest of the system. Rpager has one thread which waits for page faults, both implicit and explicit, from the Mungi kernel. The Mungi kernel tasks uses Rpager as its pager. The Mungi kernel implements the abstractions described above. The Mungi kernel has one thread to handle page faults from user threads and another thread for handling system calls. Each APD in the system is implemented using an L4 task. Mungi threads map directly on to an L4 thread.

User threads execute system calls by sending a short IPC message to the kernel's syscall thread. There are some instances such as thread creation, and exception handling, where the Mungi kernel can not directly perform the required operation, because it is not in the

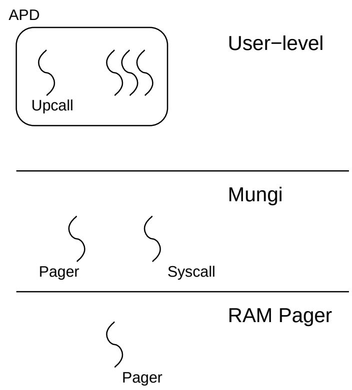

Figure 5.2: Overview of the current Mungi implementation.

same L4 task as the user thread. To solve this problem the first thread in each L4 task, the *upcall* thread, is dedicated to executing code on the kernel's behalf.

5.3 Persistence in Mungi

Periodically the Mungi system will save the state of the entire system in an operation known as a checkpoint [NBH02]. Every checkpoint saves all the data in the system including system data, and the current state of executing threads. This regular checkpointing provides the system with persistence, so that when a system is shutdown, in a controlled or uncontrolled manner, the state can be effectively restored.

The persistence system relies on a disk driver to actually write data to permanent storage. A device driver framework for Mungi must be able to be used efficiently by the persistence system.

A difficulty caused by persistence is reinitialising hardware when the system is restarted. Devices represent a lot of external state that cannot be directly saved and restored by the persistence system. There needs to be a way to notify a device driver that the system has rebooted, so it can reinitialise the hardware.

5.4 Distribution

Mungi is designed to be a distributed system. Multiple nodes within a Mungi system will be able to access the SAS using a distributed shared memory model. This is currently unimplemented, however it is an active research topic within DiSy.

Just as the persistence system was the major user of disk drivers, the distribution system is the major user of network devices. Although network devices will also be used to support legacy protocols such as TCP/IP, its primary use will be to support distribution. The design of the network interface should be focused on providing an efficient interface on which to implement distribution.

5.5 Resource management

Resource management in Mungi is based on a bank account model [HLR98]. Bank accounts are charged rent for the storage space occupied by their objects. Each bank account receives a regular income from the system. Recent work [CH02] has extended this model to account for physical memory as well as backing storage.

5.6 Mandatory access control

The Mungi system supports *mandatory access control* (MAC) [Edw00]. MAC allows a system-wide security policy to be enforced. The security policy is a user level module which is called by the kernel when validation is required. This system complements the flexible discretionary access control provided by capabilities.

5.7 Mungi naming service

Although a 64-bit address is sufficient to name any object in a Mungi system, they are not very convenient for humans to remember and use. Mungi has a flexible user level naming service which can be used to name any object in the system [Gra01]. The naming service is not restricted to naming Mungi objects; it is able to name any 64-bit value. This means that component-instances can also be easily named.

5.8 Mungi component system

The *Mungi component system* (MCS) provides a framework for the construction of secure, extensible, system level modules [EH01]. An MCS component consists of a *component class* and individual *component instances*. MCS provides the mechanism for creating and invoking methods on component instances.

Each Mungi component class executes within a separate APD, providing encapsulation from other parts of the system. The state of each component instance is stored in an object within the component class's APD. Individual instances are identified by the address of their state data. Since multiple instances can exist within a single Mungi object normal capabilities are not sufficient for access control. A valid *component instance capability* (CICAP) is required to invoke a method on an instance. This access control is implemented by the class itself, rather than the Mungi kernel. To invoke a method, the caller executes a PDX call to a registered entry point in the component class.

A PDX call is only able to pass a single 128-bit capability as an argument, however executing a method requires passing a 128-bit CICAP in addition to any arguments the method itself may require. To avoid this problem, the actual arguments, and the CICAP, are marshaled by the caller into a parameter buffer. The capability to the parameter buffer is then passed to the component class, which then needs to unmarshal the arguments and perform access control checks on the CICAP before executing the method.

The code required to marshal arguments and check CICAPs is a significant part of the MCS framework. To avoid having to write this for every new component an *interface definition language* (IDL), is defined, from which this code can be generated. MCS uses an IDL which is a subset of the Object Management Group's *common object request broker architecture* (CORBA) IDL [COR99].

5.9 MCS compiler modification

A lot of components and interfaces are used in the device driver framework. Unfortunately, there were some shortcomings with Pax, the MCS IDL compiler. The problems, and their solutions, are described in this section.

5.9.1 Sequences

The OMG CORBA IDL, on which the MCS IDL is based, provides a syntax for describing variables length lists or *sequences*. Each item in a sequence must be of the same type. Support for sequences has been added to the MCS IDL compiler, which required modification of the grammar, and changes to the code generation.

C, unlike some higher level languages, has no way of representing sequences in the core language. The chosen manner for representing sequences is as a struct:

```
struct {
   unsigned int c;
   <sequence-type> * v;
};
```

The first field, `c`, indicates the number of elements in the sequence, and `v` is a pointer to the actual items in the sequence.

When passing sequences between components the caller should ensure that the sequence data is inside the parameter buffer, so the callee is able to access it.

5.9.2 Make compatibility

The Mungi system currently uses the build tool *make*. Some of the aspects of the IDL compiler made it difficult to integrate with make.

The files generated by the IDL compiler were based on the name of the interfaces or components defined in the file, rather than the file name itself. The aspect made it very difficult to construct make targets. To avoid this problem the output files are now based on the input filename. Unfortunately, this means there is a limitation of defining only one component or interface per file, although in practice is not a large problem.

The IDL compiler produces both files that you need to edit, *stub files*, and files which do not require any changes. If the original input file is changed it is safe to overwrite files that do not require modification, however it is undesirable to overwrite stub files, which may have been modified by the user. The compiler was modified to support this preferred mode of operation. In order to allow the user, and other tools such as make, to determine which files are auto-generated the file extension was changed to `.idl.[ch]`, rather than simply `.[ch]`.

The final improvement made to the IDL compiler was to make it execute the C pre-processor directly, rather than relying on a build script.

5.9.3 Name space clashes

In order to shorten the length of identifier names produced, the IDL compiler performed a simple form of compression on component and interface names, by only treating the upper case characters as significant. For example `ISource` would be translated into `is`. The problem with this method is that the namespace is quickly reduced, and name space clashes become a real problem.

To overcome this problem the compiler has been changed so that it does not try to reduce the length of identifier names. Although this means that at times rather long identifier names are generated, this is clearly better than code that will not compile.

5.9.4 Registration map generation

Although Pax generated most of the code required to write a component, it did not generate the code to set up the component and register it with the component locator. This drawback has been rectified, and the IDL compiler now generates registration mode code and a `main` function which is compatible with the `load` command (see Appendix B.1.1).

5.10 Mungi configuration

During the course of this thesis it became necessary to improve the Mungi start up procedure. Previously all initialisation of the system was hard coded into the *main* function of the first thread to be executed. To avoid this problem I added a simple configuration to the Mungi start up procedure. A plain text configuration file is included in the DIT image and parsed by the initialisation thread. The syntax is similar to that of UNIX shell files.

A simple example configuration file is shown in Figure **??**.

```
echo Starting Mungi
# Bootstrap the component system
load clocator

# Start the naming service
load naming

# Create some initial directories
mkdir /dev
mkdir /drivers
mkdir /drivers/pci
```

Various commands will be introduced at relevant sections in the descriptions of the various device drivers. A complete documentation of the configuration commands can be found in Appendix B.

5.11 Previous Mungi device driver work

The creation of a device driver framework in Mungi has been attempted previously [Kis01], although in the more restricted context of porting Linux device drivers. This section examines the proposal and explains why it is necessary to revisit this topic.

The framework proposed by Kisliakov was that each device driver was a *Mungi component system* (MCS) component, which interacted with a system-wide resource manager object.

5.11.1 Device drivers

Each device driver component may support more than one type of device, and more than instance of each type of device. To support identification of devices inside each component the device driver creates an major and minor number structure which mirrors the way a Linux system identifies devices. The problem with this breakdown is that it introduces a separate namespace inside the device driver, which violates the main premise of a SASOS. This requires creating yet another special type of capability, a device capability. Introducing a new capability types means another system must be written do handle security checking, on top of the existing object capabilities and *component instance capabilities* (CICAP).

The user interface to the device driver, emulates that of a Linux device driver. Unfortunately, this interface implies copying data between the device driver and the user, rather than a zero-copy interface which would exploit the features of the SASOS.

5.11.2 Resource manager

The resource manager provides a way for individual device drivers to gain access to system-wide resources. The resource manager provides an interface that allows device drivers to access:

- physical memory,
- I/O regions,
- interrupts and,
- timers.

There are several problems involved in this approach. The interface defined contains many functions which are orthogonal, and there is no need to implement these functions in the same component. The major problem is that the thesis does not explain how accessing physical memory can be implemented on a Mungi system.

The proposed interrupt mechanism has one major flaw, in that it uses raw L4 abstractions for handling interrupts in Mungi user land. As the resource manager is just a normal Mungi component it should only use Mungi, not L4, mechanisms. Delivery of the interrupt from the resource manager to the device driver is done by a protection domain extension (PDX) call. The use of a PDX call seems the most logical, unfortunately the method presented presents a problem; during the PDX call the CPU interrupt is disabled[1], as a result any IRQ that shares the CPU interrupt will be masked until the PDX call returns. At best this increases interrupt latency, at worst, a malicious device driver could never return from a PDX call and effectively disable other devices in the system. This limitation of extended latency is noted later in the thesis, however the denial of service issue is not addressed.

Although the timer functions provided in the resource manager are useful, they could easily be provided by a separate timer device driver. The timer functions also suffer the same denial of service problems experienced by the interrupt interface.

5.11.3 Porting Linux drivers

The major focus of Kisliakov's thesis [Kis01] was not on a general device driver framework, rather a framework for porting Linux device drivers. The approach taken to reusing Linux device drivers was to provide a set of emulation libraries which would be linked against a Linux device driver. The thesis details the parts of the Linux kernel that are needed for a device to work and provides a general emulation library that works for the *mem* character driver, which is really just a virtual device as it does not actually access any physical hardware. One of the problems found during the porting of Linux device drivers was that they have a "tendency to not call the provided kernel functions for a particular task, and implement virtually the same functionality" themselves.

[1] This is not explained in the thesis, but is inferred from how interrupts in L4 operate

Chapter 6

Reuse of existing device drivers

There are many different hardware devices in existence, and this poses a major problem for writers of new operating systems. Although large commercial vendors, such as Microsoft, can rely on hardware manufacturers to provide drivers, and large open source projects, such as Linux, can rely on an active developer community to produce new device drivers, new operating systems do not have this luxury.

One popular way to avoid this problem is to try and reuse existing device drivers, usually from open source projects such as Linux or BSD.

6.1 OSKit

The Flux OSKit [FBB+97] project provides a set of libraries and components that implement many of the core services found in operating systems such as boot loaders, memory management, network stacks, and, of course, device drivers. By using these components the costs associated with creating new operating systems can be greatly reduced.

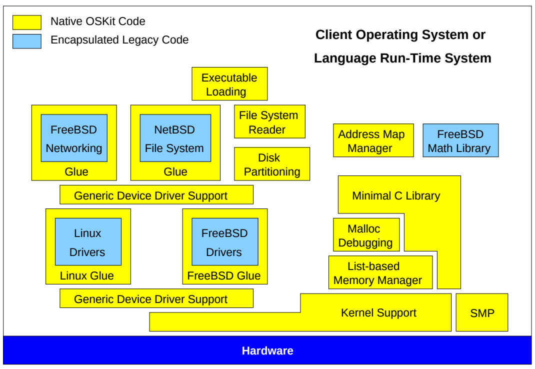

Figure 6.1: OSKit structure. Reproduced from: [FBB+97]

The device drivers provided by the OSKit framework are taken, largely unmodified, from the FreeBSD and Linux kernels. This is achieved by wrapping the device drivers in an emulation, or "glue", layer. This approach means that as the drivers are modified they can be easily included in OSKit.

6.2 Mach

The Mach project reused Linux drivers in their kernel [GD96] with a great deal of success. This project involved writing an emulation layer which allowed all network and SCSI drivers from the Linux 1.3.35 kernel to be compiled into the Mach kernel. During benchmarks they found that this emulation layer did impact on performance, however the increase in latency was minimal in most cases. Despite the overall success experienced in this project it should be noted that this work was done on an earlier, and arguably simpler, version of Linux. Also, despite being a microkernel, device drivers are implemented inside the Mach kernel, which significantly lowers the complexity and overhead involved in an emulation library.

6.3 Sawmill

Another project using existing Linux drivers is the Sawmill [GJP+00]. This project attempts to decompose Linux into separate components. Device drivers are separated into individual components which communicate with other components via IPC.

6.4 Summary

Although some of the work presented in this chapter has been positive other work such as Kisliakov's thesis [Kis01] has shown that driver reuse is not a simple thing. Recent work on the Nooks architecture [SMLE02], shows that there are at least 147 different calls made from device drivers into the kernel, and at least 103 calls made from the kernel into device drivers in the Linux kernel. This complexity is one the reasons I have chosen to avoid this path in my research. The other reason for avoiding the reuse of device drivers in Mungi, at least at this stage, is that reusing drivers from an existing operating systems limits the design of Mungi device drivers to the traditional I/O abstractions.

Chapter 7

Device Driver Domain-specific-languages

7.1 Devil

Instead of trying to isolate device drivers, the Devil [MRC+00] project attempts to reduce driver complexity, and consequently increase robustness, by defining a *domain specific language* in which to write a device driver.

Rather than directly writing code that accesses registers a developer defines what registers exist on a device and what functionality each register provides. Given this specification a compiler generates code that interfaces with the hardware.

The Devil IDL allows the driver write to specify various aspects of how a device operates. This mainly involves specifying the device's registers and their functionality. The language also allows was indicating dependencies between different registers.

The Devil project only deals with a small subset of device driver development. It does not attempt to solve other problems, such as the interaction protocol and interaction with the rest of the operating system or client interfaces.

7.2 Device-register language

It was decided at an early stage that using a tool such as Devil would make the development of device drivers in Mungi simpler, and also make the driver itself more robust.

Devil was evaluated but was found to be unsuitable because the code it generated was tailored towards both Linux and the IA-32 platform. Unfortunately, the source code to Devil was unavailable[1] making it impossible to extend the tool to work with Mungi.

As there were no other existing tools, a *device-driver domain-specific language* (DDDSL) was created. The aim of this tool is to convert a simple description of a devices' registers,

[1]The source for the compiler has since been released.

and produce a set of C functions, structures, and macros which makes it easy to access and manipulate these registers in a device driver. DDDSL was designed to be portable across different platforms and operating systems.

The features of DDDSL are best explained through use of an example. A sample input file is shown in Figure 7.1. The first two lines define some parameters used by the compiler. As discussed in previous sections the methods for accessing registers differs between platforms, and even different I/O space on the same platform; the `space` parameter indicates how to access the device registers.

Line 5 describes a register call UART_FCR, which is 8 bits in size and is offset by two bytes from the base register. The `WO` attribute indicates that this is a write only register. The following lines describe the bit fields residing inside the register.

```
;space=alpha_sparse
;handle=cthis->serial_port

/* The UART flow control register */
UART_FCR 8 @ 0x2 {WO}:
        fifo_en <0>
        rx_reset <1>
        tx_reset <2>
        dma_select <3>
        fifo_size <6:7>
```

Figure 7.1: An extract from the Mungi serial driver register description file.

The DDDSL compiler generates two output files. The first is a set of structures and unions describing each of the registers. The output of the previous example is shown in Figure 7.2. The compiler generates a bit field so that the programmer can directly access any of the register's fields without needing to explicitly use bit-shifting operations. Extra fields, such as `padding0`, represent the unused bits of the register. The generated union allows the programmer to access the register as either the raw value, or through the bit field.

```
struct uart_fcr_s {
        volatile uint8_t fifo_en : 1;
        volatile uint8_t rx_reset : 1;
        volatile uint8_t tx_reset : 1;
        volatile uint8_t dma_select : 1;
        volatile uint8_t padding0 : 2;
        volatile uint8_t fifo_size : 2;
};

union uart_fcr {
        struct uart_fcr_s r;
        uint8_t data;
};
```

Figure 7.2: C structures output by the DDDSL compiler.

The other output produced is a set of macros and functions for accessing the registers. This is shown in Figure 7.3.

```
static inline void write_uart_fcr(bus_space_t bus, union uart_fcr foo) {
        return alpha_sparse_write_8 (bus, (2), foo.data);
}

#define WRITE_UART_FCR(foo) write_uart_fcr(cthis->serial_port, foo)
```

Figure 7.3: Macros generated by DDDSL.

The `write_<register_name>` function takes two arguments: a handle to the registers, and the actual value. The handle indicates where the registers are mapped into the address space, which is generally not known at compile time. If the source file indicates a `handle` parameter the compiler also generates a `WRITE_<REGISTER_NAME>` function simply calls the `write_<register_name>` function with a default handle argument. The advantage of using this interface over simply directly accessing registers is that you have a level of type safety not usually available when interacting with device registers.

An example of how this is used in real code is shown in Figure 7.4. This example, which is taken from Mungi's serial driver, initialises the serial chip's FIFO buffer.

```
union uart_fcr uart_fcr;

/* Enable FIFO */
uart_fcr.r.fifo_en = 1;
uart_fcr.r.rx_reset = 1;
uart_fcr.r.tx_reset = 1;
uart_fcr.r.dma_select = 0;
uart_fcr.r.fifo_size = FIFO_TRIGGER_8;
WRITE_UART_FCR(uart_fcr);
```

Figure 7.4: Example usage of DDDSL generated code.

DDDSL was used during the implementation of Mungi device drivers. It has also been used successfully in the BiOS [Ste02] project.

Chapter 8

Asynchronous I/O

One of the early design decisions to be made in the design of Mungi device drivers was how the device drivers should interact with other parts of the system.

8.1 Active producers – passive consumers

In conventional operating systems, programs invoke a read or write system call when they wish to transfer data. In such systems the user program is both an active producer, and an active consumer; all data transfer is initiated by the program. in contrast, the operating system is a passive entity (Figure 8.1).

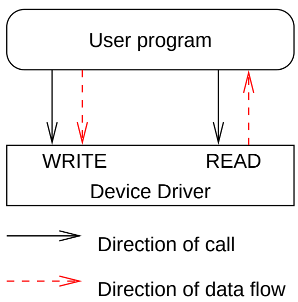

Figure 8.1: Active program – passive device driver.

Although this model of I/O is quite pervasive, it is not without difficulties. Problems occur when a program needs to read from more than one source at once, because it is only possible to read from one source at once. This mode of operation is becoming more popular in server class of applications such as Web or database servers which have many active connections at once, and also on the desktop where graphical programs need to accept data from keyboards, mice and the network stack concurrently.

One way to handle reading from multiple sources at once is to spawn a thread for each possible input. The overhead of threads causes serious degradation as the number of input streams increase [WCB01].

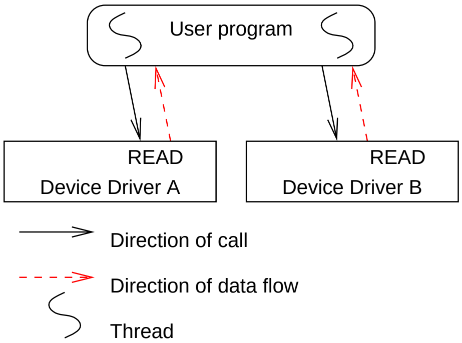

Figure 8.2: Reading from multiple inputs with threads.

Another commonly used technique to multiplex input is the `select` system call. The select system call has also been shown to scale poorly [BMD99]. Apart from this, a select-like interface would not be possible to implement if the required inputs are produced by two different modules. This is not a problem in monolithic kernel, however it would be in a Mungi system where each device driver is in a separate component.

A problem with this model is that a program can not directly transfer data to another program. Unix provides pipes as a way to connect two programs together, which has the drawback of buffering user data inside the kernel. Another disadvantage is that transferring data requires both a read and write. This model does not seem appropriate for the Mungi, where device drivers are simply components transferring data to other components.

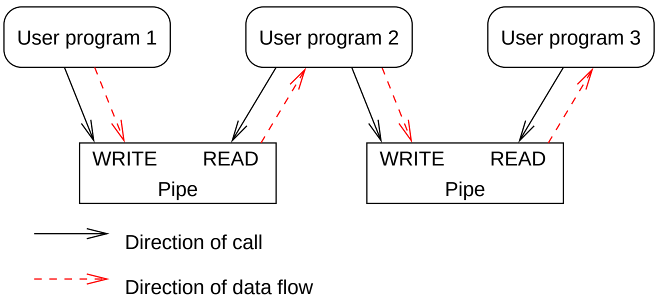

Figure 8.3: Data transfer with pipes.

An alternative mechanism is for the producer to actively write data to the consumer. This model of pushing data through the system has been used for data transfer between modules inside a kernel [Rit84], and recent work has looked at using this model for communicating between user level modules [Ste01]. In a UNIX-like system, providing this model would require an upcall [Cla85] — from the kernel — to the user program. Research into using upcalls on Linux has be shown to improve performance of web servers, by avoiding the problems associated with the traditional data transfer model [PSW01]. In Mungi there is no need to introduce special mechanism to enable this, since all device drivers run at user level anyway the existing PDX mechanism can be used.

Using this model data can be directly transferred between different components, without the need for an intermediate buffer. This is important because any unnecessary buffering

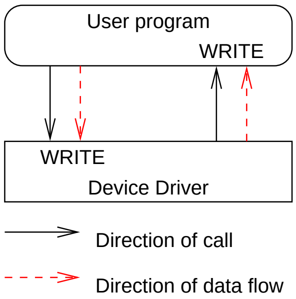

Figure 8.4: An active producer model.

is going to degrade performance.

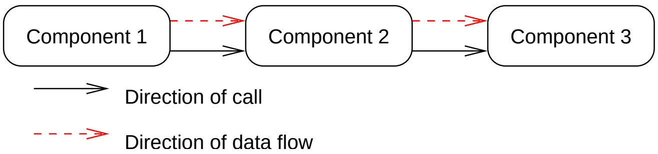

Figure 8.5: Inter-component transfer.

Accepting input from multiple sources becomes very simple in this model, as each possible input source will simply call the program when data is available. Since the program may need to differentiate between different input streams, the producer needs to identify itself when calling the program. The only difficulty with this model is that the program needs to deal with concurrency, however this is no different from the multi-threaded technique described previously.

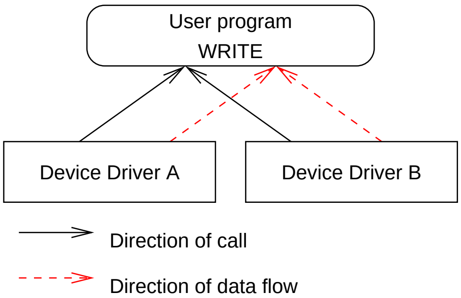

Figure 8.6: Multiple inputs in an active – producer model.

In this model it is clear that there will need to be some control functions required to indicate where the producer should be sending data. This is not a real burden on the programmer and actually offers some advantages. The connection between producers and consumer in the system could easily be controlled externally rather than inside each user program, as is the case in the traditional model. This could mean that a running component can have its input stream changed at run time, which could be very useful in a number of situations. If a network card malfunctions in a high availability server, it could be replaced at run-time, and then connected directly to the existing server application. Another use in a distributed

Mungi system would be moving an active GUI session from one node to another by simply changing the input mouse and keyboard, without requiring the session to be stopped and restarted.

8.2 Asynchronous vs. synchronous

The previous section described two different ways of performing data transfer between clients and devices, one in which the client was the only active entity, and one in which the device driver called the client program. The calls between components, in either method, can be handled in two different ways:

- the client blocks until the data is transferred, or,
- item the call returns immediately, and the data is transferred asynchronously.

8.2.1 Synchronous

An example of how blocking I/O could be implemented in Mungi is illustrated in Figure 8.7 and described here:

1. Client sends a request to the device driver with a PDX call.
2. Driver performs driver-specific action to setup and initiate data transfer.
3. Client thread blocks on a semaphore inside the device driver.
4. Actual data transfer occurs.
5. Semaphore is signalled.
6. PDX call returns.

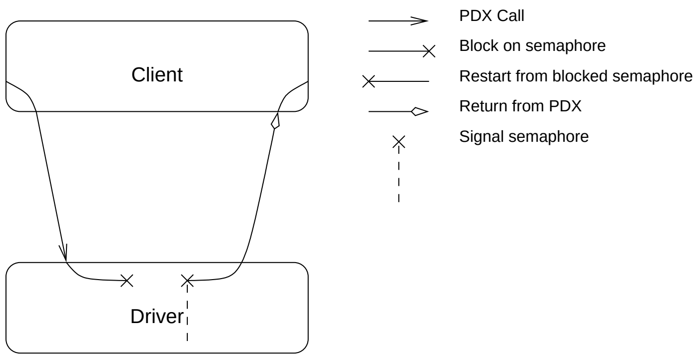

Figure 8.7: A blocking interface.

Even if a device driver exports a blocking interface, the client program can easily emulate a non-blocking interface by spawning a dedicated thread to perform the I/O operation. The main thread is then free to continue execution up until the point where it requires the result of the I/O operation, at this point it blocks on a semaphore. When the I/O thread has returned from the device driver it can signal the semaphore and then die. This operation is illustrated in Figure 8.8.

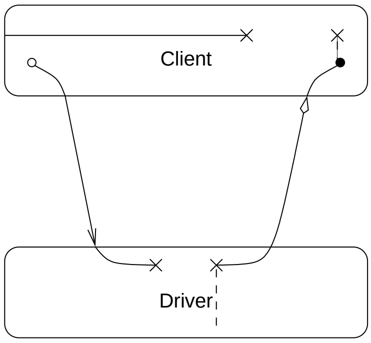

Figure 8.8: Emulation of non-blocking with a blocking interface.

The main advantage with this model is that if all I/O in the system is synchronous this will be efficient, since no special way of handling it is required. Another minor advantage is that this is a very familiar model of I/O for many programmers.

8.2.2 Asynchronous

An alternative non-blocking model is now presented. The steps involved using this method are (see also Figure 8.9):

1. Client sends a request to the device driver with a PDX call.
2. Driver performs driver-specific action to setup and initiate data transfer.
3. Initial PDX call returns.
4. Actual data transfer occurs.
5. Device driver sends the result to the client with a PDX call. (Depending on the internals of the device driver this may require the creation of a thread.)
6. Second PDX call returns.

It is possible to emulate a blocking I/O interface given this underlying non-blocking model (see Figure 8.10). After sending the initial request to the device driver, the thread blocks on a semaphore. When data transfer is completed, the callback PDX simply signals the semaphore and the main thread continues. This is very similar to the initial blocking model, except that blocking on the semaphore is now the client's responsibility rather than the driver's.

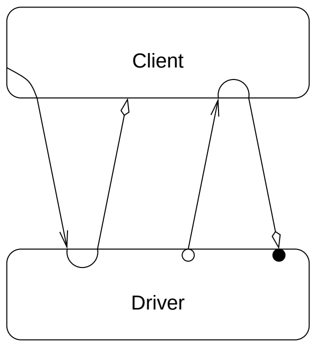

Figure 8.9: A non-blocking interface

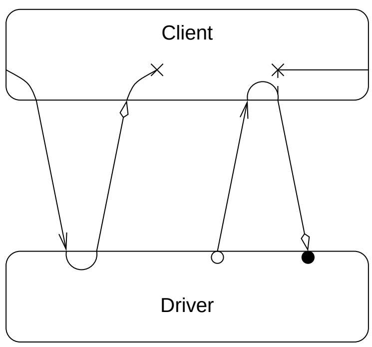

Figure 8.10: Emulation of a blocking with a non-blocking interface

Chapter 9

Design of a Mungi Device Driver Framework

Running device drivers at user-level requires the operating system to provide some mechanisms for accessing the hardware in the system. This chapter presents a design for how Mungi can provide these mechanisms.

This chapter also presents an overview of how device drivers can be built on top of these mechanisms.

9.1 Device register interface

Mapping device registers from the physical address space into Mungi's single address space (SAS) is the best method of providing drivers access to the required device registers. If the registers for each device are contained inside a Mungi object, then the existing security and protection mechanisms can be employed. This is shown in Figure 9.1.

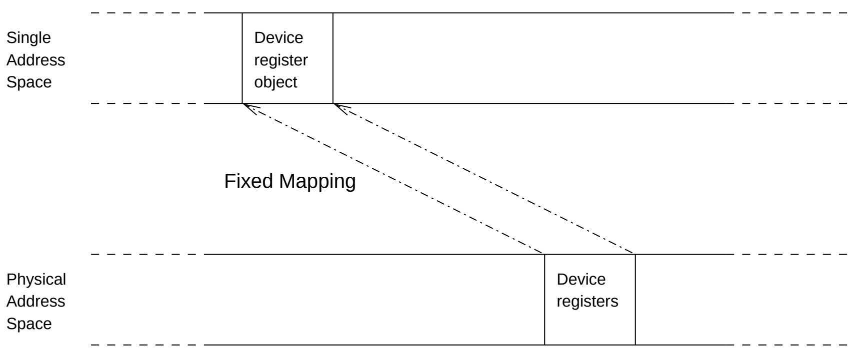

Figure 9.1: Mapping device registers into the single address space.

The exact mechanisms by which these mappings can be created must be addressed. As discussed earlier, access to device registers cannot be cached, so the establishment of these mappings must also ensure the cache is disabled.

Processor	Virtual Address Space Size	Physical Address Space Size
Alpha 21164 [Sam97]	43	40
Alpha 21264 [Com99]	48/43	43
MIPS R4700 [R4695]	40	36
Itanium [Int00b]	64	63

Table 9.1: Comparison of address space sizes.

The simplest way to achieve this would involve using the current API's page mapping interface. This would require the creation of a special object which maps the entire set of device registers into the single address space. A smaller object can then be created and have the required pages mapped into it using Mungi's PageMap system call. This makes a logical extension of the σ_0 model [ERH96]. This is illustrated in Figure 9.2.

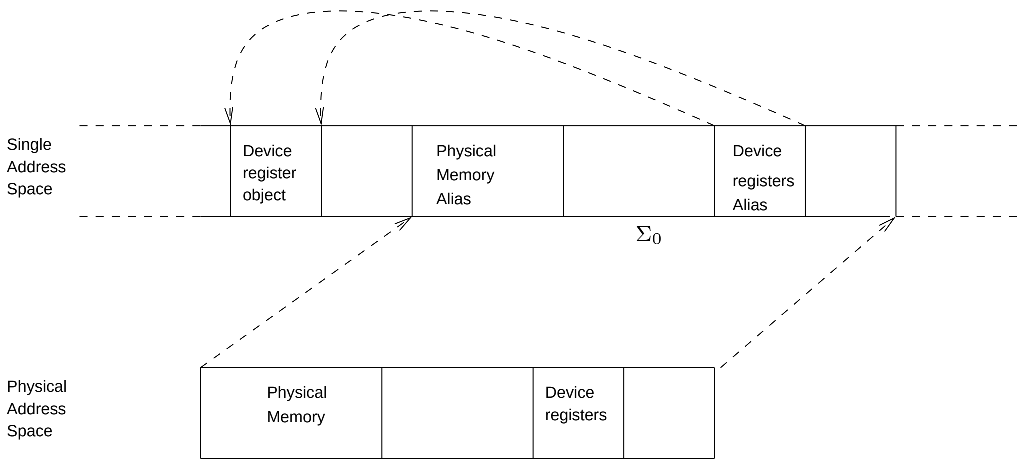

Figure 9.2: Device registers mapped using σ_0

Unfortunately there are a series of problems with this. The first is that the amount of physical memory dedicated to device registers is quite large; around 40 bits [DEC97] on the Alpha 433 workstations. This is a significant part of the SAS, especially considering that the processor only supports 43-bit virtual addresses [Sam97]. Table 9.1 shows the virtual and physical address space sizes for some 64-bit processors. Although this overhead may be acceptable on a single machine, on a distributed system each node's σ_0 area would quickly fill the entire SAS (assuming that the σ_0 area is accessible on each machine as proposed by Elphinstone).

Partition the virtual address space into local, and distributable segments would avoid the unnecessary wastage of the SAS. In such a design the σ_0 would reside within the local segment as shown in Figure 9.3. In my opinion adding this segmentation to the Mungi model seriously weakens claims to the purity of the single address space. In practice it means that many parts of the system need to explicitly consider which node they are providing services for which would not be a problem if we did not have this segmentation.

Finally, rather than adding a special memory region, we can instead introduce a new special type of object: an *I/O-space object*. Mungi already defines special objects for protection domains and bank accounts; this would simply extend the existing abstractions. On creation of an I/O-space object, the physical address of the device registers would be specified in

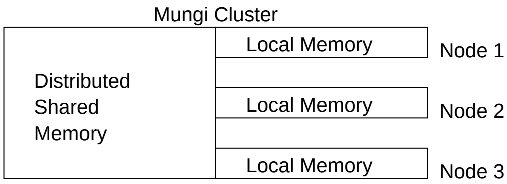

Figure 9.3: A segmented Mungi

the *user_info* field.

This introduces a security problem, because only authorised users should be able to create an I/O space object. Another problem is that in a distributed system, the physical address is not enough to uniquely identify the device registers, the node on which they reside must also be specified somehow. Both these problems can be solved by the introduction of a node object. The node object is created on system initialisation and passed to the initial user thread. To create an I/O space object a capability to the node object must be held. The node object also uniquely specifies the physical address space being referenced.

User-level policy

Both these designs only specify a mechanism for obtaining device mappings from the Mungi kernel. In both cases a privileged service which has the capability to either σ_0 or the node object will be required to implement a specific allocation policy.

Both the proposed mechanisms use the existing protection mechanisms and abstractions provided by Mungi; however the first one adds the need for segmentation and the second one requires adding a node object. The addition of segmentation to the Mungi system violates the concept a single address space, because of this the I/O space object, mechanism is chosen as the preferred design alternative.

9.2 Interrupt interface

This section proposes an interface to interrupts in a system. As discussed in Section 2.2, there are different types of interrupts that we may be interested in. This section is concerned with how CPU interrupts should be exported to user mode. The client code must be able to register interest in a CPU interrupt, be notified when an interrupt is raised and control when it wants to receive an interrupt. Although Mungi is currently implemented on top of the L4 microkernel, which has its own interface for handling interrupts, the designs examined here are not dependent on this in any way.

One possible interface for interacting with interrupts is to represent each interrupt as a pair of semaphores. One semaphore in the pair is used to indicate when the the interrupt should be enabled, while the other is used to indicate when an interrupt has occurred. So a simple interrupt routine for a device driver would be:

This model is not dissimilar from the L4 model of waiting for an IPC message from a special thread.

```
do:
      SemSignal irq_sem[0]
      SemWait   irq_sem[1]
      <interrupt occurs - do processing>
```

The alternative is for a device driver to register a PDX entry point which is then called when an interrupt occurs. The PDX parameter indicates which interrupt occurred; this is useful in the case when one function needs to handle multiple interrupts. In this model the interrupt is disabled until the PDX call returns. This technique is similar to the approach taken by the in the SunOS kernel, where a new kernel thread is created to handle an interrupt [EKB+92]. A problem with this model is how to inform the kernel of which PDX entry point to use for each interrupt; Mungi currently has no way to configure kernel parameters. This is a problem that is not limited to interrupt handling and as such is examined in a separate section.

It is difficult to determine which of these approaches is superior; neither have any inherent performance drawbacks, and either model should be equally easy to use. There are, however, two main reasons for choosing the PDX model over the semaphore model. The first reason is that the semaphore model implies long lived server threads, which is against the Mungi philosophy of providing services through PDX entry points. The second reason for choosing the PDX model is due to the potential performance benefits in a symmetric multi processing system. In the PDX model is is simple to create new PDX threads on an unloaded CPU, however in the semaphore model the interrupt thread would is already bound to a processor, and migrating it to a different processor could be costly.

9.3 Direct memory access interface

As discussed in Section 3.4, direct memory access (DMA) requires some important capabilities. Firstly, to perform a DMA transfer the device driver must be able to resolve the virtual-to-bus address mapping and secondly must be able to ensure the buffer it is performing DMA on is not swapped out during the transfer. This section examines a series of possible solutions to this problem. When evaluating different solutions, an assumption has been made that the device driver will want to perform DMA on an arbitrary buffer handed to it by a client program.

9.3.1 Pinning and physical address translation

This section examines a number of different methods for pinning memory and performing the virtual-to-physical translation.

The σ_0 model

One way to achieve this would be to use the method specified by Elphinstone et al. [ERH96]. The basic idea behind this is that there is an area of virtual memory that maps onto physical memory. A special object is then created in the area, and by using the user-level pager API, a device driver can map the memory handed to it into this object.

The advantage behind this model is that no API changes are needed and we maintain a pure SASOS.

The disadvantage is that if the page to be DMA-ed is already resident, as will almost always be the case, it will require copying the page into the special page. This heavily impacts on performance, and reduces the usefulness of unified buffering, which is one of the major advantages a SASOS has when it comes to devices drivers. It also means that the CPU is copying memory, which entirely defeats the purpose of using DMA.

Special I/O buffers

To overcome the major problem encountered in the σ_0 model, we modify it so that rather than the device driver placing the data into physical memory, the client must ensure that the buffer is in this special memory area. This model is very similar to the *fbufs* [DP93] model where all I/O uses special buffers, and suffers the same drawbacks, specifically, limiting the memory from which a client can perform I/O.

The advantage of avoiding API changes remains, however, we are able to overcome the excessive copying required in the σ_0 model.

This model has some disadvantages though. Each client will need to have rights to its own special, physical memory object. In a system with many clients, maintaining and managing these objects could become problematic. Secondly, any memory in these objects is unusable by the system as a whole, which could impact on overall performance by increasing swapping.

Another problem is that it makes I/O a special case that clients must explicitly code for. A client now has to know if the object it is interfacing with is actually a device (as opposed to a component that has a similar interface, like, for example, a loop-back network device) and whether this device needs DMA-able memory.

It is also likely that excessive memory copying will result from this model, as a client moves data back and forth between special memory and normal memory. Even this assumes that the client is aware of which parts of memory it needs to use in I/O operations, this becomes impossible if you consider a system where arbitrary pages are swapped out to disk. We conclude that a successful design must be able to pin arbitrary pages in memory.

PagePin system call

Rather than trying to use the paging mechanism in Mungi, another option would be to add two new system calls. *PagePin*, which, given a virtual address, would pin the page in memory and return the physical address at which it was residing. A corresponding *PageUnpin* call would be used when the driver has completed any DMA operation and it is safe to swap the page out.

This model avoids the problems found in the previous models; specifically it avoids any need to copy memory or restrict the pages which can be used for I/O.

One problem with this model is how to handle protection; there is no model in Mungi for restricting the use of system calls. This means that any thread would be free to pin memory, which is clearly undesirable.

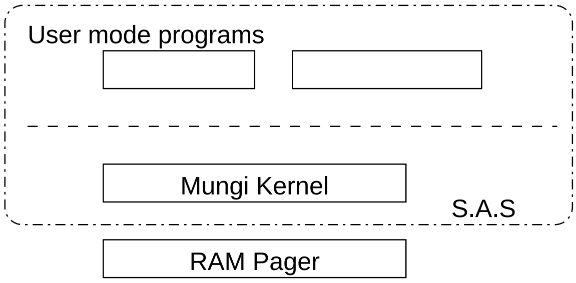

Figure 9.4: External RAM pager

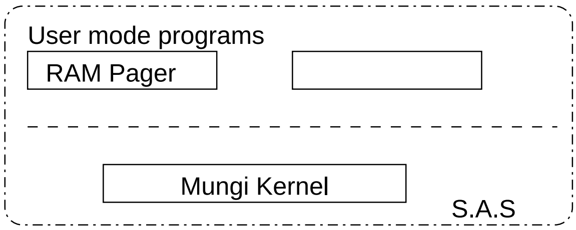

Figure 9.5: User-level RAM pager

User-level RAM pager

The problems associated with either using the paging system calls or introducing special system calls can be avoided if paging is done at user level rather than at kernel level.

In the current Mungi system, paging of physical memory is handled by an L4 task outside the single address space, see Figure 9.4. This means that any pinning logic must be handled by the Mungi kernel.

We can avoid relying on the kernel to mandate pinning policy if the Mungi RAM pager is moved from being a separate L4 task below the Mungi kernel into Mungi user mode, as shown in Figure 9.5.

In this scenario the *PagePin* and *PageUnpin* system calls as explained above would become PDX calls into the RAM pager. This avoids the problem of adding new system calls to the kernel, and integrates cleanly into existing protection models.

Moving the RAM pager from outside the single address space to being a user-level program is a major architectural change for the Mungi system. It would be too drastic a change if all it did was allow a clean way to pin and unpin pages. However, moving the RAM pager to user level also significantly simplifies other aspects of the system, specifically the interaction between the pager and device drivers. The paging system obviously needs to interact with a disk driver to allow it to swap pages in and out of memory. If the RAM pager were to run outside the SAS in L4, any interaction with a user level disk driver would be tedious; messages would need to be formatted from the pager to the Mungi kernel, and then a PDX call from the kernel into the user level driver. A similar case is required on the return path. This is a lot of indirection which implies both a performance penalty and unduly complicates the kernel; these problems are non-existent when both the pager and driver are running inside the SAS.

9.4 Driver composition

There are many different ways of structuring device drivers in a system. Usually they are simply directly compiled into the OS kernel, or are modules that can be dynamically loaded into the kernel. Other systems represent devices as user level threads, and still others treat device drivers simply as libraries that should be compiled directly into the client program. In Mungi the basic interaction between device drivers is with PDX calls. There are some other considerations required apart from this base interaction though.

One device driver in a system may support multiple physical devices. For example, systems often have more than one serial port. Obviously some method is needed to identify which physical device is being operated on when calling the device driver. UNIX-like systems solve this problem by the use of major and minor numbers. The major number identifies the device driver and the minor number identifies the specific device.

Device drivers also sometimes want to restrict access to certain interfaces, for example you may allow anyone to output to the console, but only allow certain users to read from it. In a UNIX-like system, devices are represented by files and therefore devices access is controlled with the same coarse grained access control as files.

Finally in newer systems that support hot-pluggable devices such as *universal serial bus* (USB) devices, there is a need to load drivers dynamically.

The *Mungi component system* [Edw00] already supports these features, and as a result is chosen as the basis for the Mungi device driver framework. Each device driver is represented by an MCS *component*. Each device in the system is then represented by a *component instance*. New component instances can easily be created when a new device is connected to the system. Access control is governed by the use of *component instance capabilities*.

9.5 Data transfer interfaces

The design of the different data transfer interfaces is presented in Chapters 12, 13, and 14.

9.6 Device driver control interfaces

Most hardware devices generally have more functionality than simply `read` and `write`. This functionality is hard to generalise across devices, which led UNIX-like systems to the free-form system call, `ioctl`. In UNIX-like systems, there is no defined way for a programmer to query a device to determine which functions it supports. The control interface for the Mungi device driver framework should allow the flexibility of `ioctl` but avoid its shortcomings of weakly defined interfaces.

It would be very easy for the Mungi device framework to emulate an `ioctl`-style interface — the Mungi PDX primitive essentially provides this for free. However the *Mungi component system* (MCS) [Edw00] has already solved the problem of defining structured interfaces on top of the unstructured PDX call.

Each device should export a defined control interface. It is expected that a set of standard interfaces will be defined for common devices. In the cases where a standard interface is not

```
ISerialControl {
    void SetBaud(int rate);
    void SetFlowControl(int type);
    int GetBaud();
    int GetFlowControl();
};
```

Figure 9.6: *ISerialControl* interface

available or not sufficient, additional control interfaces can be defined. There is nothing to stop a device from exporting multiple control interfaces if required. Two examples of the interface are shown in

Figures 9.6 and 9.7.

```
ICDRomControl {
    track_info * GetTrackInfo()
    void PlayTrack(int);
    void Eject();
    void Stop();
};
```

Figure 9.7: *ICDRomControl* interface

The use of MCS interfaces for defining control functions does not solve the underlying problem of many devices presenting many different interfaces, however it does allow the interfaces to be exported in a controlled manner, and for a client programmer to query a device and determine which interfaces it supports. This is a clear advantage over the simple `ioctl` design.

Chapter 10

Modification of the Mungi Kernel

This section details the changes made to the Mungi kernel including device objects, interrupt handling, page pinning and the node object. Although mostly complete the current implementation of Mungi does not have a user-level RAM pager. As a result the current implementation uses *PagePin* and *PageUnpin* system calls, which will be removed when a user-level RAM pager is implemented.

10.1 Device objects

Device objects provide the mechanism for mapping device registers into the single address space.

10.1.1 ObjCreate

The object create system call has been extended to allow the creation of device objects. An additional special object flag, `O_DEV`, is defined to identify device objects. The *pager* field of the *object info* structure is used to identify the physical address. The controlling object field must contain a valid capability to the *node object* (Section 10.4) or else creation is not allowed.

The only changes required in the kernel is checking that the controlling object capability is valid and that the controlling object points to a node object.[1]

10.1.2 Pager changes

The main change required to support device objects is in the Mungi pager and its communication with the RAM pager. When Mungi recieves a fault on a device object it sends a specially formatted message to the RAM pager indicating the physical address that should be mapped in this case. This contrasts with the normal page fault handling logic, which allows the RAM pager to choose the physical address used.

[1] The current implementation does not perform mandatory access control on the controlling object.

10.2 Interrupt handling

Interrupts are delivered to drivers as a PDX call. Two different implementations were written, and both are discussed below. How interrupts are registered with the kernel is discussed in Section 10.4.

10.2.1 Interrupts via Mungi

In this implementation, an L4 thread is created, and registered with each interrupt in the system. When an interrupt is delivered by L4, the interrupt handling code creates a PDX call into the user level handler. To ensure that the interrupt is not prematurely enabled, the interrupt thread waits until it recieves an IPC from the syscall thread, indicating that the user level handler has returned.

This implementation requires the syscall handler thread to have explicit knowledge of which PDX calls are interrupt upcalls, since when they return it needs to signal the interrupt thread. This implementation adds some additional overhead to interrupt handling, since an extra three IPCs are required to deliver the interrupt.

```
int
interrupt_handler(cap_t param)
{
        l4_threadid_t sender;
        l4_ipc_reg_msg_t msg;
        l4_msgdope_t result;

        /* Password field contains the irq. */
        int irq = param.passwd;

        /* Address field points to the handler. */
        pdx_t handler = (pdx_t) param.address;

        /* Register for interrupt with L4. */
        sender = (l4_threadid_t) {0x80 | irq};
        l4_ipc_receive(sender, L4_IPC_SHORT_MSG, &msg,
                        L4_IPC_NEVER, &result);

        sender = (l4_threadid_t) {irq};

        while(1) {
                /* Wait for interrupt from L4. */
                l4_ipc_receive(sender, L4_IPC_SHORT_MSG, &msg,
                                L4_IPC_NEVER, &result);
                /* Execute interrupt handler. */
                handler();
        }
}
```

Figure 10.1: Interrupt handler stub.

10.2.2 Interrupts directly to driver

An alternative implementation moves the thread which recieves the interrupt IPC from L4 into a user-level task. In this implementation when an interrupt is registered by the user, the kernel creates a thread in the user's active protection domain. This thread executes some stub code which waits for an interrupt IPC from L4. When an interrupt occurs, the thread can simply do a function call to the registered interrupt handler. This is shown in Figure 10.1.

By having L4 deliver interrupts directly to the user level interrupt handler the implementation is simplified and extra IPC overhead is avoided.

Both the implementations are functionally equivalent and provide the same interface to the user; both implementations were tested with no change to the user-level device driver code. This is important because the user-level device drivers should only be using Mungi abstractions – not L4 abstractions.

10.3 Page pinning

Two additional system calls (see Table 10.1) are added to Mungi to provide the mechanism required for pinning pages in memory.

System calls:	
PagePin	(*virt_addr*) → (*phys_addr*)
	fails: *protection_violation*
PageUnpin	(*virt_addr*)
	fails: *protection_violation*

Table 10.1: System calls dealing with page pinning.

One of the major shortcomings with this implementation is that there is no security, or resource control over who can execute this call. Another problem with that is that there is no effecient method of pinning multiple pages at once, which could be desirable for various drivers in the system.

Both these issues should be overcome when the RAM pager is moved to user level and a PDX interface replaces the system call interface as proposed in Section 9.3.1. Note that these system calls will be removed in the future.

10.4 Node object

The *node object* is created by Mungi when it is first booted. A a capability to the node object, is passed to the first user-level thread in the system. This changes existing startup behaviour, which provided the initial thread with a pointer to the L4 kernel information page. The node object capability confers the right to create device objects.

The node object also acts as a kernel information and configuration object. The following fields are defined:

- **l4_kip** – a pointer to the L4 kernel information page.
- **version** – the kernel version number.
- **num_irqs** – the number of CPU interrupts available in the system.
- **irq_handlers** – an array of capabilities that point to an associated interrupt handler.

Interrupt handlers can be registered with the kernel by placing a valid capability into the irq_handlers field. The user must perform a *PageFlush* on the node object for any changes to propagate to the kernel.

Chapter 11

Design and Implementation of I/O Bus Drivers

Most devices in the system do not reside directly on the system bus, instead some form of I/O bus is provided. Each I/O bus has a controller which generally provides one or more address spaces into which device registers are mapped, some form of interrupt logic, and some method of performing DMA.

This chapter describes two widely used busses: the *industry standard architecture* (ISA) bus, and the *peripheral component interconnect* (PCI) bus. A design of the MCS interfaces used to interact these busses is also described.

11.1 ISA

The *industry standard architecture* (ISA) [ASS95] bus was a popular 16-bit I/O bus which has now been superseded by the PCI bus as the primary bus in workstations. The ISA bus is usually provided as a secondary bus in most modern systems.

ISA DMA

The ISA bus uses a centralised DMA controller which must be programmed to control DMA operations. The DMA controller has four sets of registers, know as DMA channels, that specify the current operation. Devices attach to a specific DMA channel, place the transfer parameters into the registers and signal the DMA controller when they are ready to transfer data. The DMA controller always places physical addresses on the bus. ISA DMA is only able to address the first 16MB of physical memory, which is a severe limitation on current systems which have considerably more physical memory.

11.2 PCI

The *peripheral component interconnect* (PCI) bus [SA99] is a local bus architecture commonly used in many workstations and desktop computers.

The PCI bus provides 3 separate 32-bit address spaces in which device registers may be mapped. The three address spaces are:

- Configuration address space.
- I/O address space.
- Memory address space.

The method of accessing these distinct address spaces is platform dependent. On IA-32 platforms the I/O address space is mapped directly into the I/O address space provided by the architecture, the memory address space is mapped 1-1 with the physical address space, and the configuration address space is accessed through a special set of registers. The Alpha platform differs greatly and simply maps all three address spaces into different parts of its physical address space.

11.2.1 Configuration space

Each device is allocated 256 bytes of the configuration space. The format of the first 64 bytes of this space is defined by the PCI specification. These fixed registers provide enough information to the system to allow for resources to be allocated and suitable device drivers to be loaded. A description of the important registers is described below:

- **Vendor ID** register is a 16-bit value which uniquely identifies the hardware manufacturer of the device. The identifiers are allocated by a central authority.
- **Device ID** register is a 16-bit value which is assigned by the device manufacturer to identify the device. This register, combined with the vendor ID can be used to locate a suitable device driver.
- **Base Address** registers describe the regions of I/O and memory address spaced used by the device. These are configured by the system at power-on time.
- **Interrupt line** register identifies which interrupt line this device is routed to.

11.2.2 PCI direct memory access

Unlike the ISA architecture there is no centralised DMA controller on the PCI bus. Each device is able to independently perform reads and writes to physical memory. Access to the bus is controlled by a bus arbiter. When performing a DMA the PCI device places a 32-bit address on the bus. This address is generally presented directly to the memory controller.

11.3 MCS device register interface

In keeping with the rest of the Mungi design, the kernel only provides the mechanism for obtaining device objects. The Mungi kernel only provides the mechanism to map parts of the physical memory, it does not know how the PCI address spaces are mapped into physical memory, nor does it perform any resource management on the creation of device objects.

The information of how to map PCI address spaces into the physical address space is platform dependent, so this policy should be encapsulated within a specific chipset driver. Rather than directly retrieving mappings from the kernel, device drivers request specific parts of the PCI address space from the chipset driver. This allows the chipset driver to perform the necessary translation, and also perform any resource management, for example, ensuring that each section of the address space is only mapped once. The following set of MCS interfaces are defined:

- IPCI_MemSpace,
- IPCI_IOSpace and,
- IISA_IOSpace and,

These interfaces are fully documented in Appendix A.3.

Each of these interfaces exports a `space_map` method, which takes a bus address and a size parameter. `Space_map` returns the client a handle, which is described next.

The advantage of having these interposing interfaces is that allows both the kernel, and the device drivers, to be largely platform independent.

11.3.1 Register access macros

Obtaining access to the relevant registers is only half the problem for device drivers; they are also required to interact with these registers. It has been shown that the methods for accessing device register are not only platform dependent, but also depend on exactly which registers are being accessed. To solve this problem, a set of device register access function is defined for each different type of I/O space. A function is provided for reading or writing registers of various sizes. The first parameter of each function is the *handle* returned by a `space_map` method. The handle contains a capability to the device object in which the registers reside, the number of accessible registers, and an identifier indicating which space the handle is for. The appropriate set of functions can be selected either statically at compile time, or determined dynamically at run time. This design is heavily influenced by the NetBSD `bus_space` API [Dem97].

11.4 MCS interrupt interface

As with the device registers, the Mungi interface for registering interrupts is not exposed to individual device drivers. The reason for this is, once again, to avoid the need to have platform-dependent code in either the driver or the Mungi kernel.

Two separate interfaces are defined to allow device drivers to register for receiving an interrupt; *IISA_Interrupts* and *IPCI_Interrupts* (see Appendix A.4). These interfaces export a single `register` method, which takes an interrupt number and the CICAP of a component-instance to call when an interrupt occurs. The component implementing these interfaces can provide extra semantics on to the basic Mungi abstractions, such as the ability to share interrupt between multiple device drivers.

Device drivers wishing to receive interrupts must provide the *IInterrupt* interface, which exports the single `interrupt_event` method. The `interrupt_event` method is invoked when an interrupt occurs. The interrupt is masked for the duration of the call.

11.5 21174 (pyxis) core logic chip

The *21174 core logic chip* (21174) [DEC97], also known as the Pyxis chipset, provides the interface between the PCI bus, the Alpha 21164 [Sam97] processor and system memory. This section describes how interrupts are handled with the 21174, the system address space, and the operation of the 21174's scatter-gather TLB.

11.5.1 Interrupts

The 21174 provides an interface to the system interrupt controller. An *interrupt request register* (INT_REQ) is a 64-bit register, with each bit indicating whether an interrupt is pending. A device driver is able to query this register to determine which device interrupts are pending when a CPU interrupt is raised. An *interrupt mask register* (INT_MASK) is also provided which allows each interrupt to be individually masked.

ISA interrupts are not directly provided by this interface, instead there are implemented separately using two 82C59 controllers. The 82C59 controllers generate an interrupt on the primary interrupt controller. To determine which ISA interrupt occurred the device drive must issue a PCI interrupt acknowledge cycle, which returns the state of the ISA IRQs.

11.5.2 System address space

The 21164 is able to issue 40-bit physical address. The most significant bit of the physical address controls the caching attribute, if it is zero the 21164 accesses cache memory space, otherwise uncached address space is used. The RAM in the system is mapped into the lower 8GB of the cached address space. The uncached address space contains mapping of the three PCI address spaces and other *configuration/status registers* (CSRs). Table 11.1 shows the physical address map.

PCI sparse space

The Alpha architecture is only able access memory using aligned longword and quadword references, however many PCI devices require byte and word size references. This problem is solved by encoding the transaction size and offset in the lower bits of the physical address issued by the processor. This memory is referred to as sparse since 32 bytes of physical address space are mapped on to a single byte of PCI address space.

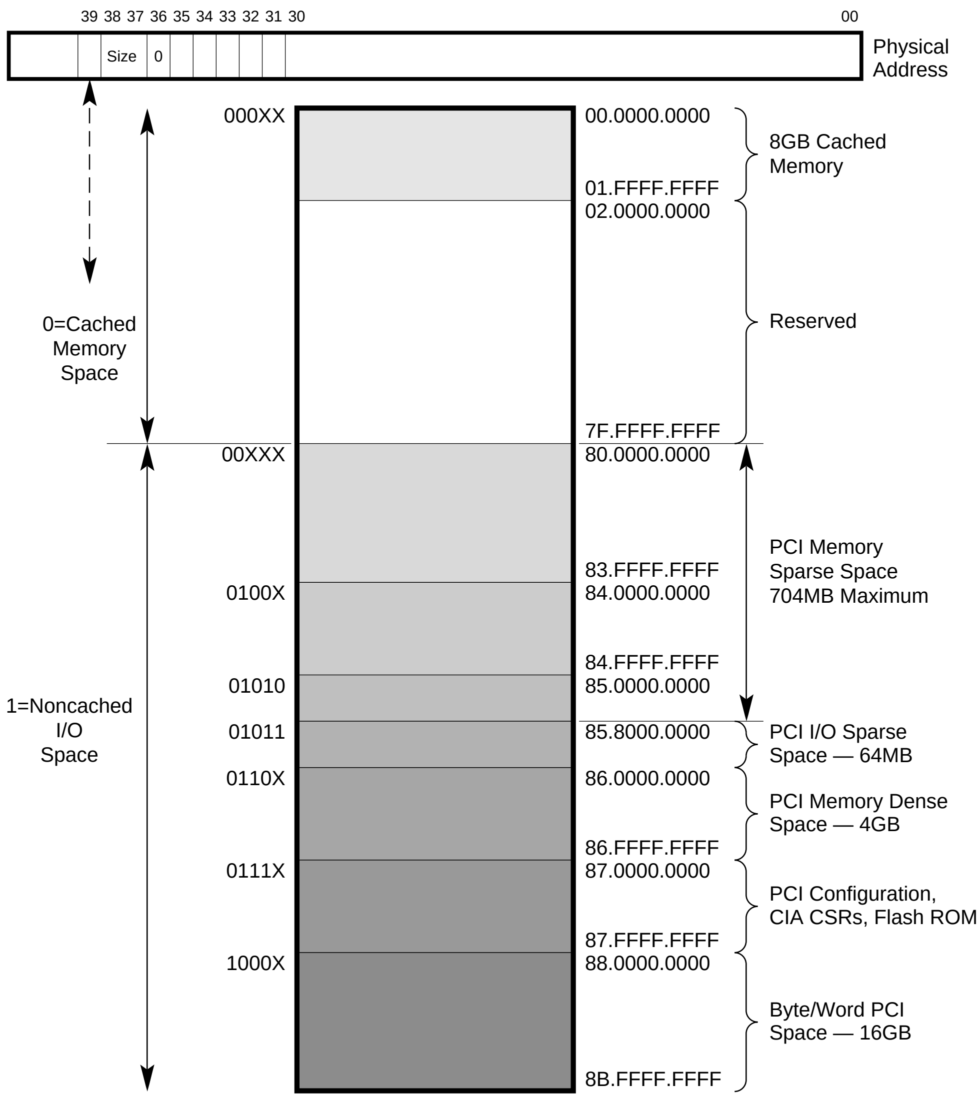

Figure 11.1: The 21174 address space overview. Reproduced from: [DEC97]

21164 address	Size(GB)	Description
00.0000.0000 – 01.FFFF.FFFF	8.00	Main memory
80.0000.0000 – 83.FFFF.FFFF	16.00	PCI sparse memory region 0, 512MB
84.0000.0000 – 84.FFFF.FFFF	4.00	PCI sparse memory region 1, 128MB
85.0000.0000 – 85.7FFF.FFFF	2.00	PCI sparse memory region 2, 64MB
85.8000.0000 – 85.BFFF.FFFF	1.00	PCI sparse I/O space region A, 32MB
85.C000.0000 – 85.FFFF.FFFF	1.00	PCI sparse I/O space region B, 32MB
86.0000.0000 – 86.FFFF.FFFF	4.00	PCI dense memory
87.0000.0000 – 87.1FFF.FFFF	0.50	PCI sparse configuration space
87.2000.0000 – 87.3FFF.FFFF	0.50	PCI special/interrupt acknowledge
87.4000.0000 – 87.4FFF.FFFF	0.25	21174 main CSRs
87.5000.0000 – 87.5FFF.FFFF	0.25	21174 memory control CSRs
87.6000.0000 – 87.6FFF.FFFF	0.25	Reserved
87.8000.0000 – 87.8FFF.FFFF	0.25	21174 miscellaneous CSRs
87.9000.0000 – 87.9FFF.FFFF	0.25	21174 power management CSRs
87.A000.0000 – 87.AFFF.FFFF	0.25	21174 interrupt control CSRs
87.B000.0000 – 87.FFFF.FFFF	1.25	Reserved

Table 11.1: 21174 physical address space map. Reproduced from [DEC97]

The 21174 maps all three PCI address spaces using this sparse space technique. To map all 32-bit of the I/O and memory address space would require an excessive amount of physical address space. To avoid this, only a subset 64MB of the I/O space is. This is a valid tradeoff, since IA-32 machines can only address 64KB of I/O space, and the use of I/O space is discouraged in the PCI specification. The first 32MB of the I/O space maps onto the lower 32MB of the I/O address space. The remaining 32MB is can be relocated to any section of the I/O space.

There are three separate memory spaces which can be used to map up to 704MB of the PCI memory address space. These three regions can mapped to different sections of the the PCI address space.

PCI dense memory

Not all devices require byte level their memory address space. A video card frame buffer, for example, can be treated as normal memory. In these cases it is more efficient to access the memory using native long-word memory accesses. The 21174 provides a direct mapping of the 4GB PCI memory address space for this purpose. PCI dense space has the advantage that PCI address are mapped to contiguous locations and it is more efficient than sparse space.

11.5.3 Scatter-gather mappings

The 21174 maps addresses presented on the PCI bus, by a device attempting to initiate a DMA transfer, to the 21164 cached memory space. This mapping is controlled by a set of *address windows*.

Each window describes a mapping from PCI address to physical memory, which can be either a direct mapping or a scatter-gather mapping.

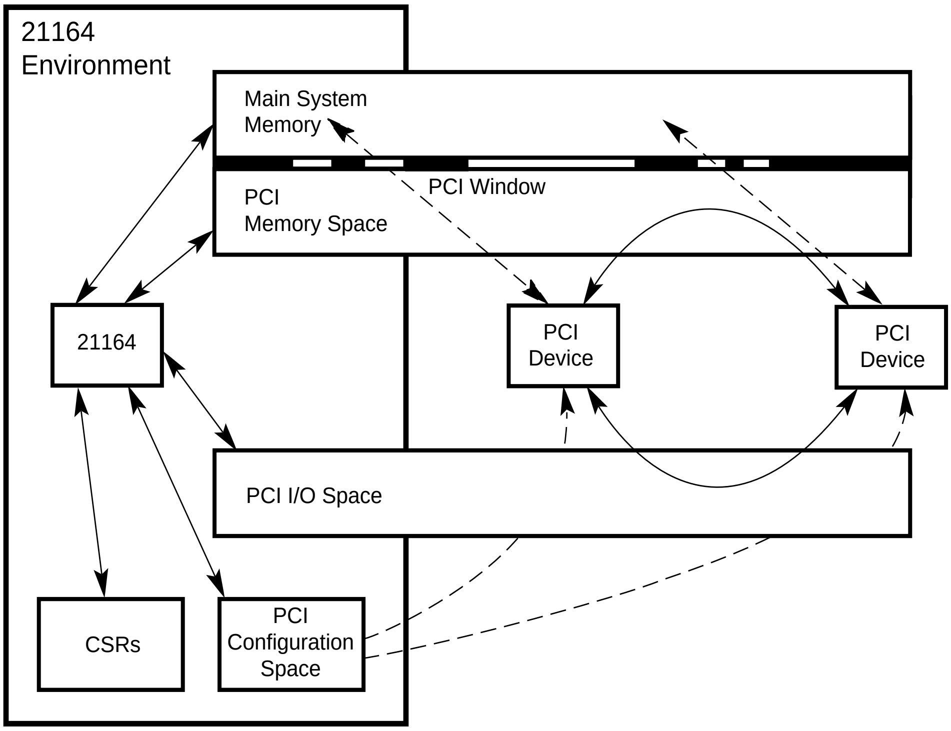

Figure 11.2: The 21174 address space overview. Reproduced from: [DEC97]

If a scatter-gather mapping is specified the 21174 converts PCI addresses to a physical address by looking up the physical address in a scatter-gather page table. These lookups are cached using a scatter-gather TLB.

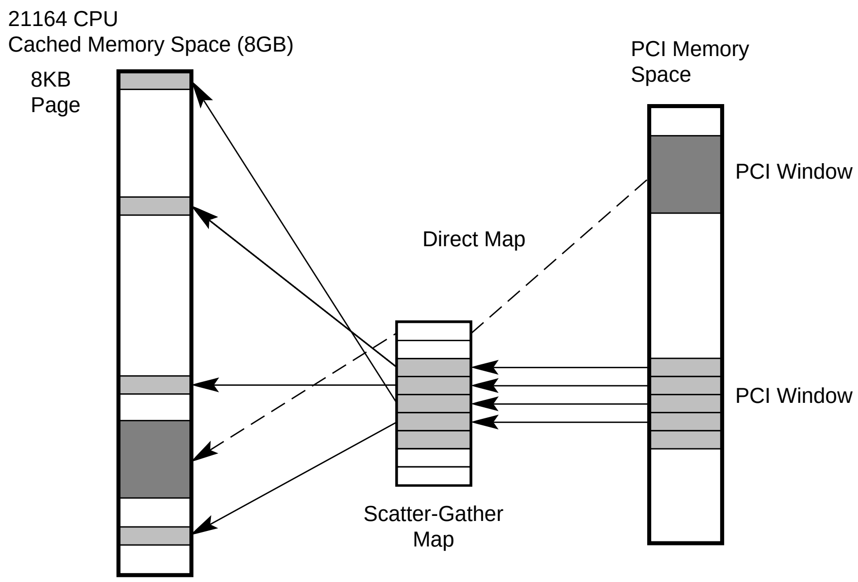

Figure 11.3: 21174 PCI window operation. Reproduced from: [DEC97]

11.6 Pyxis device driver implementation

This section describes the implementation of the Pyxis chipset driver and the interfaces that it provides.

The Pyxis chipset provides the PCI interface for the Alpha Miata platform. The ISA bus is a secondary bus and as such is also managed by the Pyxis driver. The has a set of interfaces for allocating device objects for drivers. It also provides the interface for registering interrupts.

11.6.1 IMainChip

The main chip interface (Figure 11.5) should be provided by the device driver implementing the core functionality, such as providing device objects and interrupts. When the system starts, the initialisation process calls the main chip interface's init method, providing it with the node object capability. Any component providing the main chip interface must be trusted. The main chip component can be created by using the `initcs` (see Appendix B.2.1) command.

On initialisation the Pyxis device driver registers itself for the CPU interrupts, sets up device objects so it can access its command and status registers, and sets up the device.

11.6.2 I/O and Memory Spaces

The Pyxis chip provides three different interfaces for obtaining device mappings.

- **IPCI_IOSpace**

```
#include "stdpax.idlh"
#include "imainchip.idl"

#include "ipci_iospace.idl"
#include "ipci_memspace.idl"
#include "ipci_confspace.idl"
#include "ipci_confspacerw.idl"
#include "ipci_interrupts.idl"

#include "iisa_iospace.idl"
#include "iisa_interrupts.idl"

[
 cid( 0x76 ),
 ver( 1, 1 )
]
component CPyxis {
    provides IMainChip;

    provides IPCI_ConfSpace;
    provides IPCI_ConfSpaceRW;
    provides IPCI_MemSpace;
    provides IPCI_IOSpace;
    provides IPCI_Interrupts;

    provides IISA_IOSpace;
    provides IISA_Interrupts;
};
```

Figure 11.4: Pyxis IDL definition

- **IPCI_MemSpace**
- **IISA_IOSpace**

The IDL description for `IISA_IOSpace` is shown in Figure 11.6.2. This is representative of the other interfaces. On the Alpha platform, the ISA I/O space is the same as the PCI I/O space. Separate interfaces are provided because it is concievable that this is not the case on all platforms, and that it may be desirable to protect each of these interfaces individually.

The component maps the requested address to its location in the physical address space and performs the underlying calls to `ObjCreate`. The component also ensures that the same part of the address space is not requested by multiple clients. Allocation is done on a first come, first served basis.

The component should try to ensure that the client only receives access to the requested range of registers. This is generally not a problem in the memory spaces where mappings are sparsely allocated. The ISA I/O space, however, is densely packed, and granularity is at best 256 bytes[1]. In the case where two different devices' registers share the same 256

[1]256 byte granularity is possible due to the sparse space. As each byte of I/O space takes 32 bytes of physical address space, only 256 bytes of I/O space fit in an 8kb page.

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x64 )
]
interface IMainChip {
    int init (cap_t node_cap);
};
```

Figure 11.5: Main chip IDL definition

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x67 )
]
interface IISA_IOSpace {
    bus_space_t space_map (bus_addr_t address, size_t size, int flags);
};
```

Figure 11.6: ISA I/O space IDL definition

byte region, usability wins over security and both drivers are given access to the page.

11.6.3 PCI Configuration Space

A procedural interface to the PCI configuration is provided rather than a simple memory mapping to a device driver. One reason for this is that on most platforms it is possible for two devices' configuration registers to reside in the same page. Another reason is that on most platforms, including Alpha, accessing the PCI configuration space requires setting various parameters on the chipset. Allowing all device to access these registers would violate the security of the system. The downside of this is that accessing a single register will incur a protection crossing overhead. This should be acceptable since configuration registers only need to be accessed infrequently.

Unfortunately, providing a secure procedural interface is not straightforward. An initial implementation providing the following interface:

```
uint32_t read_register(pci_device_t device, int register);
void write_register(pci_device_t device, int register, uint32_t value);
```

The problem with this simplistic interface is that it does not provide security; there is nothing to stop a malicious driver from manufacturing a bogus device parameter. One way to solve this would be to introduce a new capability, for example *PCIDeviceCap*. This is not desirable though, since it would require writing a new set of validation code.

To solve this problem a new component, *CPCIConf*, is introduced. Each CPCIConf instance provides a secure wrapper for accessing the PCI configuration space. Drivers only

```
pyxis_handler()
{
        int_req = READ_INT_REQ();

        /* Unmask all the set interrupts */
        int_mask = READ_INT_MASK();
        int_mask.data &= ~int_req.data;
        WRITE_INT_MASK(int_mask);

        /* Ack all the requested interrupts */
        WRITE_INT_REQ(int_req);

        while(int_req.data) {
                i = ffz(~int_req); /* Find the first bit set */
                int_req &= int_req - 1;

                thread_create_local(irq_spawn, i, 1);
        }

        return;
}
```

Figure 11.7: Pyxis interrupt handler

have access to their specific CPCIConf instance through a CICAP. CPCIConf instances are able to access the *IPCI_ConfSpaceRW* interface, which provides the actual hardware access. CPCIConf instances are only created by the Pyxis component are are made available through the *IPCI_ConfSpace* interface.

11.6.4 Interrupts

The Pyxis component handles interrupt for both PCI devices and ISA devices. It provides the *IPCI_Interrupts* and *IISA_Interrupts* interfaces. The interfaces allow device drivers to register for a specified interrupt. Interrupts are allocated on a first come first served basis. Devices that register for interrupts must export the *IInterrupt* interface. When an interrupt occurs the device drivers `interrupt_event` method is called.

The interrupt logic in the Pyxis component has to decode incoming CPU interrupts into device interrupts and then dispatch these to the registered device drivers. It is possible for multiple device interrupts to be coalesced into one CPU interrupt. A simplified version of the interrupt handler is shown in Figure 11.7. The interrupt handler loops through finding each interrupt that was triggered, and spawning a thread to handle the interrupt. The newly spawned thread simply calls the `interrupt_event` method in the registered driver. When the device driver returns it reenables the interrupt.

11.7 Device configuration

Device configuration refers to the process of creating a device driver and how the device driver is allocated the various resources it requires such as memory mappings and interrupts.

Some kernels take the approach that the device driver is in the best position to determine which resources it require. However, with the plug and play nature of PCI this is no longer the case because, the system is able to determine a device's requirements by examining the standard configuration registers. In the Mungi driver framework, rather than each driver requesting the resources it needs, it is provided with these resources by the system at initialisation time. This means that the framework registers any required interrupts on behalf of the device driver.

This approach also has the advantage of totally avoiding any resource management problems.

To enable this automatic configuration, each device must export a generic initialisation interface. These are common to each class of device. The ISA and PCI initialisation interfaces are described below.

11.7.1 PCI initialisation

The *IPCI_Device* interface (see Appendix A.5.1) should be provided by any PCI device driver. The interface exports the `init` method which is called by the driver framework to initialise the device. The driver is passed a CICAP to a CPCIConf instance, as well as a list of device mappings, and interrupts.

The device framework is able to determine which PCI devices are in the system and the resources they require by scanning the PCI configuration space.

For each device found during the scan an appropriate driver is found and initialised. Currently finding an appropriate driver is performed by a simple lookup. The Mungi naming service [Gra01] is used as a database for this purpose. A `/drivers/pci/vendor_id/device_id` entry, which links to the appropriate driver, is set up for each supported device in the system. This `scanpci` command performs this operation.

11.7.2 ISA initialisation

The*IISA_Device* interface (see Appendix A.5.2) is very similar to the IPCI_Device, the only difference being that the `init` method is not passed a CPCIConf CICAP.

The major difference between PCI and ISA initialisation is that there is no way to dynamically discover the devices present in the system. It is sometimes possible for ISA devices to configure themselves, but this generally requires providing access to a wide range of the I/O address space, and all the ISA interrupts. Due to these problems, rather than trying to auto-configure devices, they must be explicitly loaded, and explicitly provided with the I/O space and interrupt resources they require.

Chapter 12

Design and Implementation of a Stream Device Driver

This chapter presents an MCS interface for stream based data transfer. The implementation of a stream based device, a serial port, is also presented.

12.1 Stream device interface

Stream-like devices are those that either input or output a simple stream of bytes. This is equivalent to characters device in UNIX.

An asynchronous, active producer – passive consumer, zero-copy model is used for interfacing with stream-like devices. Three distinct interfaces are provided, *ISource*, *ISink* and *IIOComplete*.

The ISource and IIOComplete interfaces should be implemented by device drivers, or other components, that produce data. The interface is fully described in Appendix A.7.1.

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x70 )
]
interface ISource {
    int set_sink (cicap_t sink);
};
```

Figure 12.1: ISource interface.

The `set_sink` method indicates to which component, known as a *sink*, incoming data should be transferred. If the source already has a sink when `set_sink` is called, it should be changed to the newly specified sink.

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x270 )
]
interface IIOComplete {
    int complete (cicap_t from, uintptr_t key);
};
```

Figure 12.2: IIOComplete interface.

The `complete` method is the callback used to indicate the completion of an asynchronous write call. The `key` parameter indicates which write call the callback is associated with.

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x71 )
]
interface ISink {
    int feed (cicap_t notify, buf_seq data);
};
```

Figure 12.3: ISource interface.

The ISink interface is implemented by components that need to receive, or consume, data. The `notify` CICAP identifies which component sent the data. The final parameter is a variable length sequence `buf_t` structures. The layout of the `buf_t` structure is shown below.

```
typedef struct {
    cap_t cap;
    uint64_t length;
    uintptr_t key;
} buf_t;
```

Figure 12.4: The buffer structure

The `cap` field is a capability pointing to a data buffer. The `length` field indicates the size of the buffer in bytes. The `key` field is passed as a parameter the `complete` callback method described previously. The callback should only be invoked if a non-zero key is provided. By providing a sequence of `buf_t` entries on each call a set of non-contiguous regions of memory can be transferred efficiently.

Figure 12.5 shows an example `buf_t` sequence, with two individual entries. When the component has processed the second buffer the caller's `complete` method will be called with a parameter of 37. If the caller needed individual notification when each buffer is completed a `key` could be specified for each entry in the sequence.

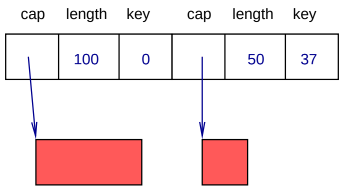

Figure 12.5: An example buffer sequence.

A component may implement either *ISource*, *ISink*, or both. A printer port for example may only provide the *ISink* interface, whereas a serial port would provide both, and a keyboard would only provide the *ISource interface*.

12.2 UART Chip

The *universal asynchronous receiver transmitter* (UART) chip [Nat95] provides an interface to serial-line communications. Although there are a variety of UARTs the most prevalent type, and the one used in the target platform, is the 16550.

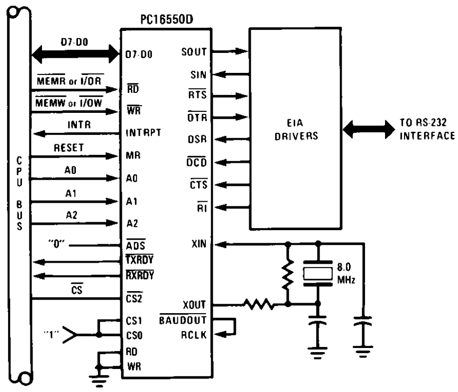

Figure 12.6: Basic configuration of a 16550 UART. Reproduced from: [Nat95]

The UART has two FIFO buffers: a transmit FIFO and a receive FIFO. The FIFOs are small, 16 bytes, which means that despite its relatively slow speeds it needs interrupts to be handled reasonably quickly to avoid overflowing the buffer.

The UART allows for various communication parameters, including baud rate, word size and parity to be set by the device driver.

UARTs' registers are usually mapped into the ISA I/O space at fixed addresses, and use fixed ISA interrupts. The first UART in a system is usually mapped at 0x3f8, and uses interrupt 4. The second UART is mapped at 0x2f8 and uses interrupt 3.

The 16550 UART is not capable of DMA, and hence all data transfer is done using programmed I/O.

12.3 Serial driver

A simple serial driver was developed during the course of the thesis. The component definition is shown in Figure 12.8.

```
#include "stdpax.idlh"
#include "interrupt.idl"
#include "iserial.idl"
#include "isource.idl"
#include "isink.idl"
#include "iisa_device.idl"
#include "iiocomplete.idl"

[
 cid(0x78),
 ver(1, 1)
]
component CSerial {
        provides IISA_Device;
        provides IInterrupt;

        provides ISerial;
        provides ISink;
        provides ISource;
        provides IIOComplete;
};
```

Figure 12.7: Serial driver component definition.

The UART is a bi-directional device, so the device driver implements both the *ISink* and *ISource* interfaces described previously. The serial driver also provides the device specific *ISerial* interface. Currently this only allows the baud to be set, however this could be extended in the future to change other settings such as the type of parity used.

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x72 )
]
interface ISerial {
    int set_baud (int rate);
};
```

Figure 12.8: Serial driver specific interface.

12.3.1 Sending data

When another component needs to send data over the serial line the `feed` method is called. The `feed` method adds an entry to the send queue. Each entry in the send queue stores a capability to the supplied data buffer, a CICAP pointing to the component that should be called when the data is transferred, a key which is passed as a parameter to the callback function, and some values used to keep track of how much of the buffer has been transferred.

```
struct write_buf {
    cap_t buffer;
    int cur_pos;
    int extent;
    cicap_t notify;
    uintptr_t key;
    struct write_buf * next;
};
```

Figure 12.9: The C struct defining an entry in the send queue.

If the send queue already has entries in it, then the `feed` method simply returns to the user, however if the send queue was previously empty it needs to initiate data transfer. The private `do_write` function is used to transfer data from the send queue to the UART. The `do_write` function attempts to writes data to the UART until either the UART's transmit FIFO is filled, or there is no more data to transfer.

The UART generates an interrupt when the transmit FIFO empties. When this happens the `do_write` function is called by the interrupt handler, allowing it to continue sending data.

When `do_write` finishes transferring a buffer it calls the IO completion callback method on the supplied component instance. A new thread is spawned to perform the callback so as to avoid the possible denial of service that would happen if the callback function never returns.

12.3.2 Receiving data

The UART generates a receive interrupt when, either the receive FIFO reaches a trigger point (usually 14 bytes), or when there is data in the FIFO and no new data has been received for a certain length of time.

On a receive interrupt the interrupt handler copies the data from the UART's FIFO into a circular buffer. If a valid *sink* has been registered the serial driver will then call the registered components `feed` method. As with the callback method, this is done by creating a new thread to protect against possibly malicious components. The region of the circular buffer passed to the user is not reused until the `sink` performs the I/O completion callback. If the circular buffer fills up then subsequent data is lost.

Chapter 13

Design and Implementation of a Packet Device Driver

This chapter presents an MCS interface for packet data transfer. The implementation of a packet based device, an Ethernet adapter, is also presented.

13.1 Packet device interface

The packet interface is very similar to the stream interface described in the previous chapter, the main difference being that rather than sending a list of buffers, a list of packets is sent. The two interfaces, *IPacketSource* (Figure 13.1) and *IPacketSink* (Figure 13.3), operate in the same manner as *IPacket* and *ISource*. The *IIOComplete* interface is used to notify the caller of I/O completion.

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x147 )
]
interface IPacketSource {
    int register_packet_sink (cicap_t sink, int filter);
};
```

Figure 13.1: IPacketSource interface.

The *IPacketSource* exports the single `register_packet_sink` method. The first argument is a CICAP to a component instance implementing the *IPacketSink* interface. The `filter` is a component specific argument which specifies which packets the registered sink should receive. For example an Ethernet driver may demultiplex different packets to different protocol stacks, as shown in Figure 13.2.

The *IPacketSink* interface provides a single `feed` method. This operates the same way as the *ISink* interface described previously, except that rather than a sequence of buffers, a

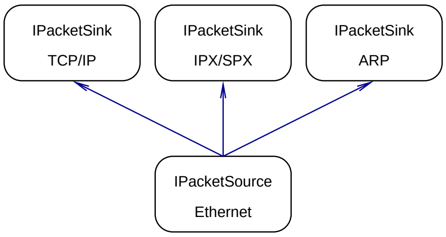

Figure 13.2: A single packet source demultiplexing to multiple packet sinks.

sequence of packets is transferred.

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x148 )
]
interface IPacketSink {
    int feed_packet (cicap_t notify, pkt_seq packets);
};
```

Figure 13.3: IPacketSink interface.

Each packet is simply a sequence of buffers. This should allow for efficient implementation of layered protocol stacks, because each layer can add headers and trailers without needing to copy data.

13.2 Ethernet driver

A driver for the DEC 21142 Ethernet chipset was developed. Although the driver is functional, it does not take full advantage of the hardware, and as a result performance is not ideal. The current implementation is however able to interact with a rudimentary TCP/IP stack developed by Matthew Chapman, another DiSy student. With this stack Mungi was able to correctly receive and send address resolution protocol packets, and reply to simple ping requests. This operation shows that the network driver is operational, and it should be possible to run a TCP based application, however due to time constraints this has not been tested.

Chapter 14

Design and Implementation of a Block Device Driver

This chapter presents an MCS interface for block based data transfer. The implementation of a block based device, an IDE hard drive, is also presented.

14.1 Block device interface

Block devices, such as hard drives and other storage devices are significantly different from stream and packet devices, and as a result, the active produce – passive consumer model is not appropriate.

The first difference is that when interacting with block devices the required data is explicitly addressed, unlike stream and packet devices where the data is simply a linear stream.

The other major difference is that storage devices, unlike say serial lines or network cards, do not independently generate data. Any incoming data from a storage device is due to an explicit request from the driver.

The *IBlock* interface provides `read` and `write` methods, which both take a CICAP and a sequence of blocks. The CICAP points to a component with an *IIOComplete* interface. This component is called when the I/O is complete.

The sequence of blocks describes the data transfer. Each block contains a capability to a data buffer, a block identifier, the number of blocks to transfer, and a key which is passed as a parameter to the I/O completion callback.

The `get_block_count` and `get_block_size` methods allow other components to determine the number and size of blocks respectively.

This interface provides a low level procedural interface to block devices. It is designed to be used by other low-level components such as the backing store filesystem. It is expected that higher level components would provide a user-level pager interface so that most user programs can simply treat it as a region of virtual memory.

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x271 )
]
interface IBlock {
        int write (cicap_t notify, block_seq data);
        int read (cicap_t notify, block_seq data);
        int get_block_count ();
        int get_block_size ();
};
```

Figure 14.1: Block device interface definition.

```
typedef struct {
    cap_t cap;
    uint64_t blockid;
    uint64_t block_count;
    uintptr_t key;
} block_t;
```

Figure 14.2: Definition of the block type.

14.2 AT Attachment

AT Attachment (ATA) is a way of interfacing host systems and storage devices, typically hard disks. It is an ANSI standard, and the latest version is specified in the *AT Attachment with Packet Interface - 7* (ATA/ATAPI-7) standard document [McL02a, McL02b][1]. The ATA interface is more commonly known as the *integrated drive electronics* (IDE) interface.

The specification defines four different protocols for transfer of data between the storage device and the host; *programmed I/O* (PIO), *single-word DMA* (SDMA), *multi-word DMA* (MDMA) and *Ultra DMA* (UDMA). Each protocol operates at a set of different cycle rates. Table 14.1 shows the different transfer rates for various ATA protocols.

The specification defines a number of commands for controlling disk drives. Some of the commonly used ones are described below.

- **IDENTIFY DEVICE** allows the device driver to retrieve information from the device. The information returned includes simple identification data such as the make, model and serial number of the drive. More importantly it also returns which features the drive supports, that speeds at which the drive can operate and the number of sectors available.

- **SET FEATURES** allows the device driver to set various drive parameters including the transfer protocol and speed.

[1] ATA/ATAPI-7 is the latest working draft. The latest official standard is ATA/ATAPI-5

Mode	Cycle Time (ns)	Max. Transfer Rate (MB/s)
PIO 0	600	3.3
PIO 1	383	5.2
PIO 2	240	8.3
PIO 3	180	11.1
PIO 4	120	16.7
MDMA 0	480	4.2
MDMA 1	150	13.3
MDMA 2	120	16.7
UDMA 0	240	16.7
UDMA 1	160	25.0
UDAM 2	120	33.3
UDMA 3	90	44.4
UDMA 4	60	66.7
UDMA 5	40	100.0
UDMA 6[a]	30	133.3

[a]UDMA 6 is currently not specified by the standard, although it is implemented by a number of drives and controllers.

Table 14.1: ATA protocol speeds.

- **READ/WRITE SECTOR(S)** transfers data between the disk and host using the PIO protocol. A 28-bit *linear block address* (LBA) specifies which sector on disk to start the transfer from. An 8-bit sector count specifies the size of the transfers.
- **READ/WRITE DMA** uses one of the DMA protocols to transfer data. The exact DMA mode is set using the `set features` command.

The ATA standard defines a set of optional feature sets. Some of the more relevant ones are examined below.

48-bit addressing

ATA is usually limited by using 28-bit sector address, which, with 512 byte sectors, allows a maximum of 128GB. Newer drives are capable of storing more than this and the ATA specification has grown to accommodate this by offering 48-bit addressing mode which allows sizes up to 144 peta-bytes. The other advantage of 48-bit addressing is that it allows sector counts to be specified as a 16-bit value, rather than an 8-bit value. This allows up to 32MB to be transferred in one operation, rather than only 128KB. To support 48-bit addressing two new commands, READ DMA EXT and WRITE DMA EXT are defined.

Queued feature set

The standard ATA specification only allows one command to execute on the device at a time. The optional queued feature set addresses this problem by allowing multiple commands, each identified by a tag, to be issued to the disk at once. This feature gives the disk drive the opportunity to optimise performance by re-ordering commands in the queue. To use this feature a device driver issues the READ/WRITE DMA QUEUED (or READ/WRITE DMA QUEUED EXT) command.

The many other optional feature sets are beyond the scope of this thesis.

PCI interface

Although the ATA/ATAPI-7 standard doesn't provide details on how the specification should be implemented, some de-facto standards have emerged. The PCI IDE controller specification [PCI94] describes how the ATA registers should be mapped into the PCI address space.

Before executing a DMA transfer command the driver must setup a *physical region descriptor* (PRD) table [Hos94]. Each PRD in the table specifies a physical base address and byte count. Each entry can specify a maximum of 64KB of memory. Multiple PRDs may be used in a single transfer, which means that non-contiguous areas of physical memory can be efficiently written to a contiguous range of disk blocks. The table is stored in the host's memory and maintained by the device driver. The controller is given a pointer to this table.

14.3 IDE device driver

The first problem involved in writing the disk controller device driver is that each controller can support up to 4 individual block devices. This is a problem because MCS provides no way to provide more than one of each type of interface on a single instance. This is similar to the problem experienced in providing the *IPCIConf* interface in the Pyxis device driver (see Section 11.6.3).

A similar solution is used. The IDE device driver exports an interface similar to *IBlock*, except that the first argument specifies which device to perform the operation on. This interface, *IIDE_Private*, is shown in Figure 14.4. For each individual disk a "glue" component is created, which provides the *IBlock* interface by simply forwarding calls to the IDE device driver. Other components in the system are not given access the the *IIDE_Private* interface.

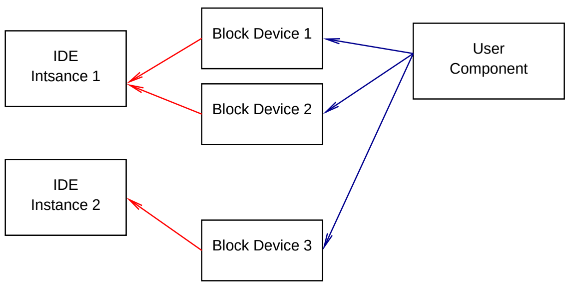

Figure 14.3: *IBlock* interface provided through indirection.

The IDE component provides *IIDE*, a device specific interface. This interface allows other components to find out how many block devices the controller currently has, and to retrieve a CICAP to these devices.

```
#include "stdpax.idlh"
#include "mungi.idlh"
[
 iid( 0x113 )
]
interface IIDE_Private {
        int write (int diskid, cicap_t notify, block_seq data);
        int read (int diskid, cicap_t notify, block_seq data);
        int get_block_count (int diskid);
};
```

Figure 14.4: The private IDE interface definition.

```
#include "stdpax.idlh"
#include "interrupt.idl"
#include "iide.idl"
#include "iide_private.idl"
#include "ifilesystem.idl"
#include "ipci_device.idl"
[
 cid( 0x89),
 ver(1, 1)
]
component CIDE {
        provides IIDE;
        provides IIDE_Private;
        provides IPCI_Device;
        provides IInterrupt;
};
```

Figure 14.5: The IDE controllers component definition.

```
#include "stdpax.idlh"
#include "mungi.idlh"
[
 iid( 0x83 )
]
interface IIDE {
        int get_disk_count ();
        cicap_t get_disk ( int diskid );
};
```

Figure 14.6: The IDE interface definition.

14.3.1 Basic implementation

The overall operation of the IDE device driver is fairly straight forward. When either the `read` or `write` method is executed the driver converts the request into a series of disk operations. Each of these disk operations is added to a queue of pending operations (see Figure 14.7).

If the queue was previously empty the first command in the queue is sent to the disk. From this point the operation of the driver is purely interrupt driven. When an interrupt occurs the driver finishes processing the current operation, and then issues the next operation to the disk. If the disk operation completes a client request then the client I/O completion callback is executed by the driver.

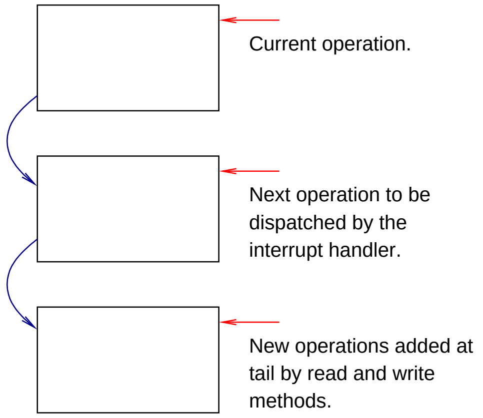

Figure 14.7: A queue of pending operations in the IDE driver.

Chapter 15

Performance

The chapter analyses the performance of device driver framework. The first two sections measure different aspects of the interrupt performance. The last section investigates the throughput performance of the IDE device driver. All the benchmarks in this chapter were taken on the Alpha Miata machine. The exact hardware configuration of this machine is described in Appendix C. The performance of the Linux operating system was measured as a comparison. All Linux benchmarks were taken using version 2.4.19, the latest stable version available at the time of writing.

15.1 Interrupts dispatching time

The first measurement taken is how long it takes to dispatch an interrupt. This provides an indication of how quickly a second interrupt can be dispatched after the first interrupt is taken.

This benchmark only measures the time spent in Mungi; it does not account for the L4 overhead. The time is measured from immediately after the interrupt IPC is received from L4, to the time when the interrupt handler waits for a new IPC. The time is measured using the Alpha `rpcc` operation, which returns the cycle count.

15.1.1 Performance with a single interrupt source

The first test measured the overhead of dispatching a PCI interrupt. Interrupts were generated by running the `hdparm` command, which reads 64MB of data from the hard drive. 10,000 interrupts were sampled. The total time spend in the Pyxis interrupt handler was measured. To gain an idea as to where the costs lay, the time taken to create the handler thread, and the time taken to acquire the lock on the interrupt mask register was also measured. Table 15.1 presents these results.

The major feature of these results is the large maximum. Further testing established that this was due interaction with the scheduler. There are times when the interrupt handler is pre-empted by another running thread. This was verified by disabling the timer interrupt

	Max. μs	Min. μs	$\overline{x}$ μs	σ_{n-1} μs
Total time	96.1	5.02	7.33	2.56
Thread creation time	92.7	2.37	4.25	1.68
Lock acquisition	1.84	0.63	0.72	0.11

Table 15.1: Time taken to dispatch interrupt with 1 active PCI device.

	Max. μs	Min. μs	$\overline{x}$ μs	σ_{n-1} μs
Total time	155	2.81	9.83	5.87
Thread creation time	111	0.99	6.08	1.30
Lock acquisition	30.6	0.61	1.06	0.90

Table 15.2: Time taken to dispatch interrupts with 2 active PCI devices.

inside the handler thread, and re-testing. Although the interrupt thread runs at a high-priority, threads created using the *ThreadCreate* system call inherit this priority. Future Mungi work should provide an interface for scheduling and prioritising threads.

15.1.2 Performance with a multiple interrupt sources

To determine how the interrupt handlers work under a heavier load, the test was re-run with activity on two independent disks. Each disk resides on a separate controller, so two different PCI interrupts are produced. These results are presented in Table 15.2.

Increasing the number of active interrupt sources in the system slightly degrades interrupt performance. The source of the extra cost is mainly in an increased thread creation cost. Comprehensive thread creation benchmarking was performed by Baumann, and showed a much smaller increase in thread creation time as the number of created threads increased [Bau02]. The conditions in the interrupt handler are slightly different from those measured by Baumann. In this case we have other active threads in the system, whereas in the tests run by Baumann newly created thread simply did a sleep call. It is possible that this increase in time is because there is more contention on the cache.

The maximum thread creation time also increases. This is expected, because, as there are more active threads in the system the cost of a pre-emption increases. There was also an increased time in the maximum cost of acquiring the interrupt mask lock, which is expected because there is more contention on it.

15.1.3 ISA interrupt performance

The dispatch path for ISA interrupts is slightly longer than that of the PCI interrupt path because handling the ISA interrupts requires accessing a separate interrupt logic chip. ISA interrupts were generated by transferring data on the serial port. A sample of 1000 interrupts was taken.

The large difference in dispatch time between PCI and ISA interrupts is not surprising, because the ISA interrupts require extra processing. The direct cost of this extra processing

	Max. μs	Min. μs	$\overline{x}$ μs	σ_{n-1} μs
Total time	156	19.8	23.9	4.65

Table 15.3: Time taken to dispatch ISA interrupts.

	Max. μs	Min. μs	$\overline{x}$ μs	σ_{n-1} μs
Single interrupt source	27.6	00.83	5.27	2.09
Multiple interrupt sources	367	1.15	7.58	12.95

Table 15.4: Time with interrupts disabled in Linux

is, on average, 9.3 μs. The remaining overhead can be attributed to indirect costs, such as cache dirtying.

15.1.4 Linux times

As a comparison similar timings were taken on the same machine running Linux 2.4.19. The times were again taken using the `rpcc` operation. The timer was started when the kernel interrupt handler is invoked. The timer is stopped when interrupts are first reenabled, either explicitly by the driver, or implicitly when the interrupt handler function returns. This timing does not include the overhead involved in dispatching the interrupt from the firmware into Linux. Again the sample size was 2000. The Linux `hdparm` program was used to read data from the hard disk, and hence generate PCI interrupts. The first test read data from one hard disk, which generated one PCI interrupt. The second test read from two drives concurrently, causing two different PCI interrupt source to generate interrupts. These results are shown in Table 15.4.

15.1.5 Comparison and validity

Mungi's results for this test are in the same order as those found on Linux. Although this is encouraging, it is not entirely fair because Linux does some actual servicing of the interrupt before re-enabling interrupts.

Although the Mungi times do not include the overhead created by L4, it is expected that these times are in the same order as the overhead from the standard firmware interrupt dispatcher used by Linux. Ideally this assumption should be tested, however I was unable to determine a suitable way to do this in L4, and the source for the standard firmware was not available.

15.2 Interrupt latency

The other important factor in interrupt performance is how long it takes to actually reach the interrupt handler after an interrupt is generated.

	Max. μs	Min. μs	$\overline{x}$ μs	σ_{n-1} μs
Single interrupt source	6290	32.2	40.1	62.8
MCS overhead	127	21.6	25.1	1.93
Multiple interrupt sources	8230	22.0	77.5	138.9
MCS overhead	179	13.1	30.1	12.6

Table 15.5: Mungi interrupt latency times.

	Max. μs	Min. μs	$\overline{x}$ μs	σ_{n-1} μs
Single interrupt source	313	0.95	5.87	13.7
Multiple interrupt sources	416	1.64	9.35	20.9

Table 15.6: Linux interrupt latency times.

15.2.1 Mungi interrupt latency

To measure this the interrupt latency in Mungi, the IDE driver was instrumented to measure how long it took to be invoked after an interrupt was generated. The actual time measured is from when the L4 interrupt IPC is received until the first line of the IDE interrupt handler. Although this measurement includes all the overhead in the Mungi system, including the Mungi component system overhead, it does not include the time taken to deliver the interrupt IPC from L4 to Mungi.

A second measurement was taken to determine the PDX and MCS overhead. This measure the time from just before the dispatch thread calls the drivers handler, to the first line of the handler.

As with the interrupt dispatching test was run once with just one disk active, and a second time with both disks active. The results are presented in Table 15.5.

The major feature of this test is the presence of some very, large maximums. Unfortunately an exact cause was not determined, although it seems likely that it is due to some pathological scheduling case.

15.2.2 Linux interrupt latency

A similar methodology was used for collecting timing measurements from Linux. These results are displayed in Table 15.6.

It is interesting to note that Linux also has some large maximum values. Unfortunately I am not familiar enough with the Linux kernel to determine what the cause of these maximums may be.

15.2.3 Comparison

The Mungi interrupt latency times were around eight times slower than that Linux. The majority of this time can be attributed to the overhead cause by MCS, including the required protection domain crossing.

15.3 IDE performance

The performance of Mungi's IDE device driver is measured and compared to that of Linux. Two different disk are used for benchmarking: a 8GB Quantum Fireball CR and a 160GB Maxtor D540X.

The test involved reading a linear set of blocks from the disk. The test was run for different data sizes from 64kb up to 64MB in powers of 2. Three different tests were run. Firstly each of the disks were benchmarked separately, and then both disks were benchmarked simultaneously. In Mungi the time was measured using the `l4_sys_time` call which returns the number of μ-seconds since boot. One Linux the `gettimeofday` system call was used, which also has μ-second accuracy.

Figure 15.1 shows the performance of IDE devices in Mungi. For longer transfers the Mungi was able to sustain a transfer rate of 34Mb/s to the Maxtor driver. The Quantum drive was able to transfer at 15MB/s. With both disks active the Maxtor's performance dropped to 26MB/s, while the Quantum drive was still able to sustain 15MB/s. The Maxtor has a buffer-cache of 2MB. When the size of data transfer drops below this the results become more erratic as the cache comes into play. A similar effect is seen on the Quantum drive, however its buffer is only 512kb so the effect doesn't occur until data transfer size is 256kb. The most interesting result spike at 256kb transfer for the Quantum drive. It is not clear why this spike occurs, as it is not present in the Linux results which use the same drive.

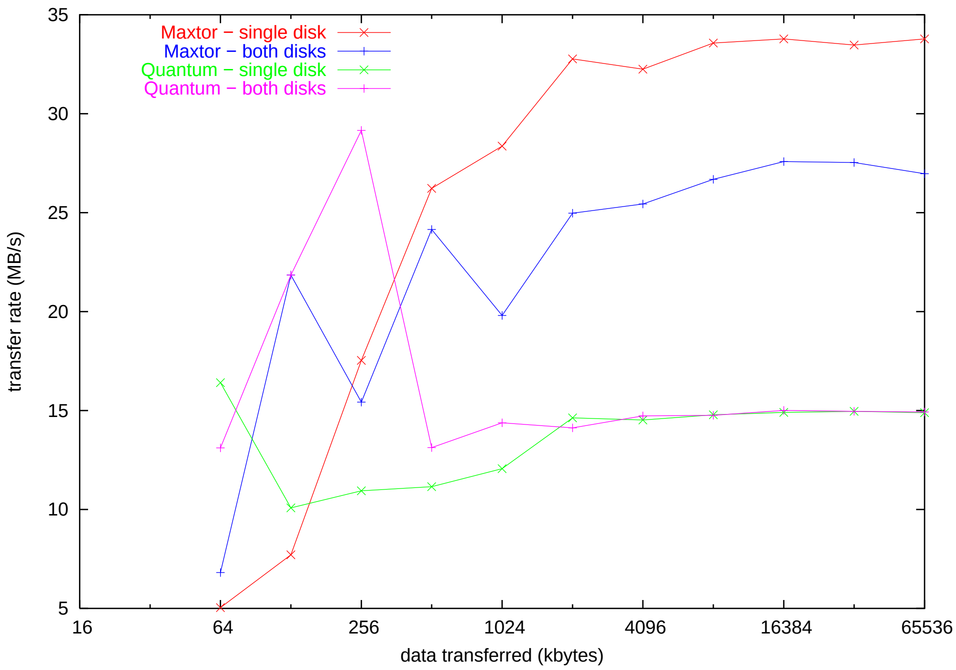

Figure 15.1: Graph showing IDE performance in Mungi.

Figure 15.2 shows the results obtained on Linux, using the same hardware.

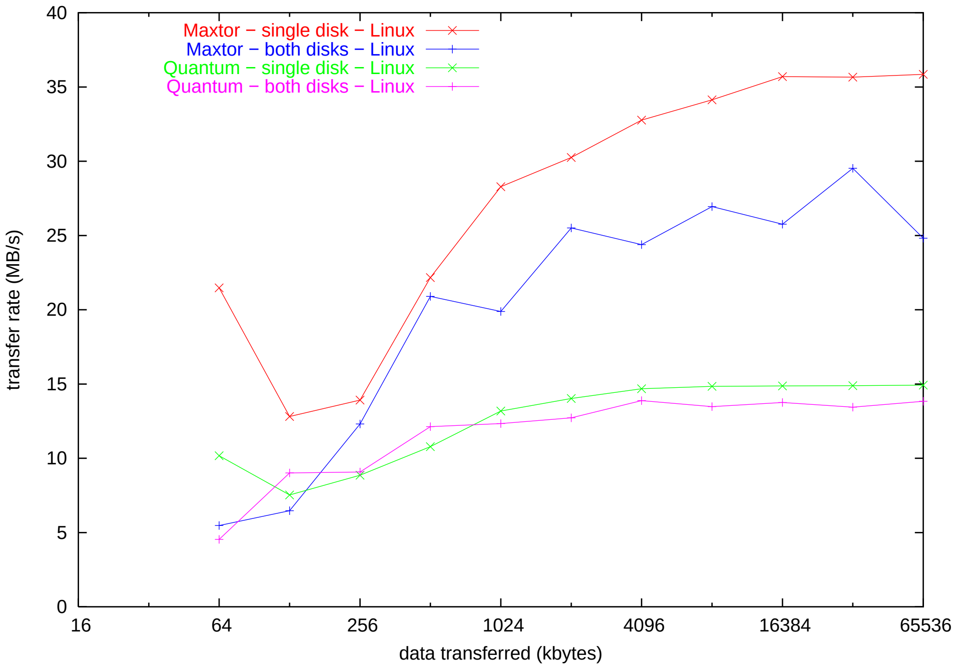

Figure 15.2: Graph showing IDE performance in Linux.

Figure 15.3 plots the performance for the Maxtor drive, under both Linux and Mungi. This plot shows that with a single active disk in the system Mungi performance is slightly worse than that of Linux, however when there are two concurrently active disks the difference in performance between Linux and Mungi becomes negligible.

Finally, Figure 15.4 shows the performance of the Quantum drive under both Linux and Mungi. When only the Quantum drive is active in the system both Linux and Mungi perform at the same rate. The major difference is that when there is concurrent disk access in the system, Linux's performance degrades whereas Mungi's does not.

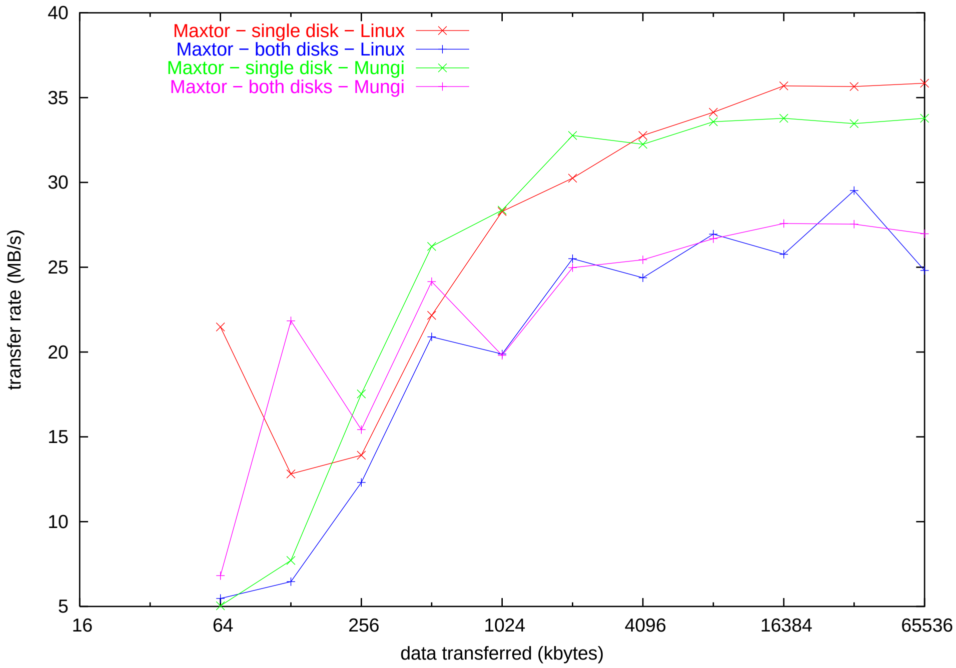

Figure 15.3: Graph showing performance of the Maxtor drive in Linux and Mungi.

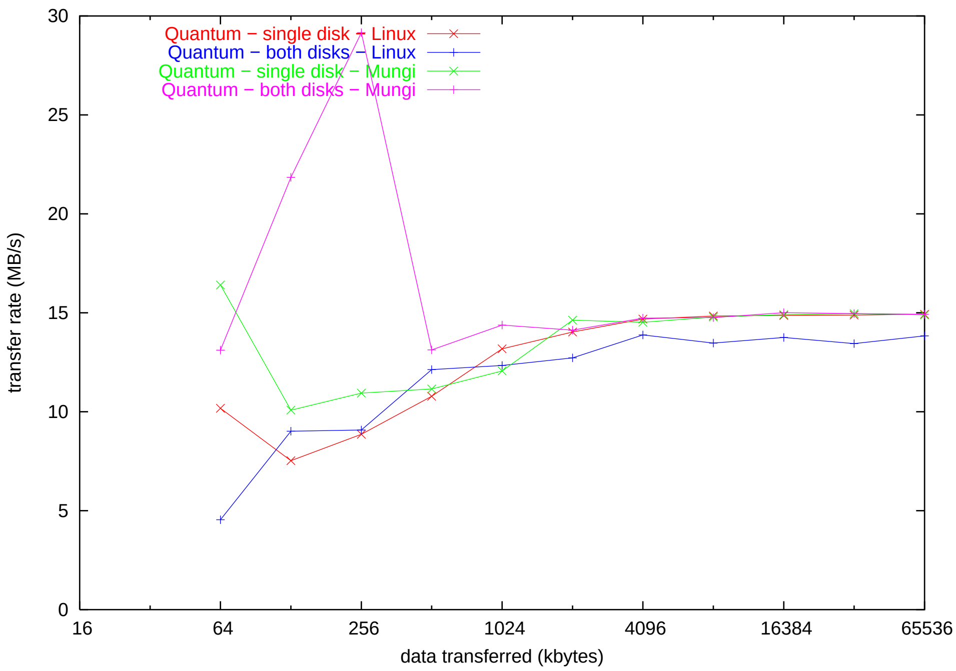

Figure 15.4: Graph showing performance of the Quantum drive in Linux and Mungi.

Chapter 16

Discussion

This chapter examines the performance achieved, and what can be done to improve the performance in the future. Some aspects of the hardware made it difficult to fully isolate device drivers. Some possible solutions to these problems are discussed. Finally, this chapter also discusses some of the difficulties encountered during the implementation.

16.1 L4 Interrupt delivery

A lot of the overhead found in the interrupt path is because Mungi must decode the CPU interrupt and then dispatch the device handler. Although the code to do this is quite short, and simple, the extra level of indirection is detrimental to performance.

This level of indirection could be avoided if, upon receiving a CPU interrupt, the L4 kernel determined the device interrupt raised, and directly invoked the device driver interrupt handler.

16.2 Hardware documentation

When implementing a device driver it is necessary to know exactly what functionality a device has, and how to correctly interact with it. There are four possibilities for obtaining this information:

Specification: Some devices, such as the IDE controller, are described in official standards. The specification of the standard usually provides enough information to create a device driver. The other advantage of programming to the specification is that the device driver should work for all devices following the specification. Unfortunately very few hardware interfaces are codified in standards, and even when this is the case, manufacturers do not always follow it to precisely.

Documentation: The easiest to use, and most authoritative source for this information is documentation supplied by the manufacturer. Most of the devices, especially the ones manufactured by Digital, had excellent documentation which made programming device drivers much easier. Unfortunately, not all manufacturers will make their documentation publicly available, which greatly impedes the development of device drivers. Another problem is that there are sometimes "quirks" in the hardware, which means that device does not work exactly as documented.

Existing source code: An other source of information comes from existing source code. The various open source operating systems such as Linux and NetBSD have device drivers for many devices. It is usually possible to determine how a device works by reading existing device driver code.

Reverse engineering: If all else fails, the final option is to determine the functionality through reverse engineering, however this process is usually slow and error prone.

16.3 Hardware

Current hardware is not ideal because it does not provide enough mechanisms to isolate devices from each other.

This is most obvious in the close packing of device registers in the ISA I/O space. It is not possible to individually protect these registers when protection is at page sized granularity. Fortunately this problem is eliminated by the PCI architecture, where the system can control, to a certain extent, where device registers are mapped.

The PCI architecture introduces another problem. Every device is capable of performing direct memory access. Although an I/O MMU can solve this problem to some extent it does not make any distinction between different devices on the bus, which means it is still possible for a device to perform unwanted DMA.

One possible solution to this would be to extend the functionality of PCI bridges. The PCI bridge provides secondary busses by forwarding transactions between the primary and secondary bus. The bridge provides a filtering function by only forwarding transactions in a certain address range on to the secondary bus. More security could be obtained if the PCI bridge could also be programmed to filter transaction originating on the secondary bus. This arrangement would mean that software could effectively treat each bus as a different protection domain.

16.4 Limitations

Unfortunately, not every aspect of the design could be implemented. These limitations are listed below and can be followed up in future work.

- `ObjCreate` does not perform mandatory access control on the node object when creating a device node.
- The capability returned from the `space_map` function is the owner cap, it should only be a read/write cap.
- The IDL compiler does not currently generate CICAP validation code, which means the is really no security on any components in the system. This bug was not fixed as it

would introduce another difficulty; the management of CICAPs. Both these problems are left as future work.

- The PCI specification provides the facility for multi-function devices. This facility has not been tested.
- A suitable protocol for restarting device drivers in the persistence system has not been designed.

Chapter 17

Conclusion

> I'm not interested in making device drivers look like user-level. They aren't, they shouldn't be, and microkernels are just stupid.
>
> *(Linus Torvalds)*

A device driver framework for Mungi has been designed and an initial prototype implemented. The usefulness of this framework has been shown by the implementation of a serial, network and IDE driver.

The performance of the IDE driver was evaluated and proved to be comparable to that of the Linux operating system.

This thesis has shown that running device drivers at user-level provides a flexibility and level of isolation that is not possible when device drivers are run inside the kernel. It has also shown that this can be done without having a detrimental effect on performance. This performance is achievable due to the abstractions provided by Mungi. Specifically the single-address-space which allows data to easily be shared between protection domains, and the PDX mechanism which provides fast cross-domain invocation.

17.0.1 Secondary achievements

Although not directly related to device drivers some other important Mungi development work was completed during the course of the thesis, specifically:

- A method for loading components dynamically.
- Changes to the Pax compiler to integrate it with the Mungi build system.
- A boot time configuration system.

17.1 Future work

Although a lot was accomplished in this thesis, there is a lot of scope for other related work, which can build on the current implementation. Some of these ideas are presented below.

17.1.1 More drivers

This thesis has only presented the implementation of a small number of devices, however there are still many more devices that Mungi is unable to use. Some immediately useful ones would be mouse driver, sound card driver and a USB driver.

17.1.2 More platforms

The current device driver implementation is only for the Alpha platform. Although the drivers were designed and implemented with portability in mind, this really needs to be tested by running the device drivers on other platforms. The one device driver that is obviously not portable is the main chipset driver, which provides the PCI and ISA interfaces. Creating the equivalent driver for other platforms would be the first priority in porting. Currently the device driver domain-specific-language only produces little-endian code, depending on the architecture this tool would also need to be extended.

17.1.3 More applications

There are currently very few application using the device driver framework. To truly test how useful these drivers are more applications will need to be written.

17.1.4 POSIX Emulation

One easy way to obtain more applications for Mungi is to port existing POSIX based programs. To allow for this a POSIX emulation layer, that convert from the native asynchronous I/O model, into the blocking I/O model will be required.

17.1.5 Integration with the persistence system

The true test of disk driver performance will only be known when it is integrated with Mungi's persistence system. This is currently under active development within the DiSy group.

17.1.6 Checkpointing device drivers

Section 5.3 described the problems involved in restarting device drivers in a persistent system. Due to time constraints this problem was not addressed in this thesis. The design and implementation of a restart protocol is required to ensure that drivers will work correctly in a persistent Mungi system.

17.1.7 Resource accounting

There is currently no resource accounting performed by any of the Mungi device drivers. One consequence of this is that it is possible for a device driver to be monopolised by one client. There is a lot of scope for future research into integrating Mungi's bank account resource management into the I/O system. Another related area of research involves providing quality of service guarantees for various devices.

17.1.8 Code reuse

Although this thesis has shown that the construction of device drivers from scratch is feasible it would be nice to leverage the existing device drive code from other open source projects such as Linux, NetBSD or OSKit. With the basic Mungi abstractions for device registers, interrupt and DMA in place, the development of some kind of emulation library or even automatic code transformation may be applicable.

Appendix A

Device Driver Interfaces

This chapter documents all the interfaces and methods in the standard device driver framework.

A.1 Miscellaneous

A.1.1 IMainChip

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x64 )
]
interface IMainChip {
    int init (cap_t node_cap);
};
```

init

SYNOPSIS

```
#include <imainchip_io.idl.h>

int init(cap_t node_cap);
```

DESCRIPTION

This `init` method should be called on start up, before any other methods on the instance. The `node_cap` should be a valid capability to a node object.

RETURN VALUE

On success, 0 is returned.

A.2 PCI Configuration space interfaces

A.2.1 IPCI_ConfSpace

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x99 )
]
interface IPCI_ConfSpace {
    cicap_t get_conf (int bus, int device, int function);
};
```

get_conf

SYNOPSIS

```
#include <ipci_confspace_io.idl.h>

cicap_t get_conf(int bus, int device, int function);
```

DESCRIPTION

The `get_conf` command is used to obtain access to the configuration space of the specified PCI device.

RETURN VALUE

On success a valid CICAP to a *CPCIConf* instance is returned. If an error occurs, or the specified device does not exist, the `ref` field of the CICAP will be NULL.

A.2.2 IPCI_ConfSpaceRW

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x107 )
]
interface IPCI_ConfSpaceRW {
    uint32_t read_32 (int bus, int device, int function, int register_num);
    void write_32 (int bus, int device, int function, int register_num, uint32_t value);
    uint16_t read_16 (int bus, int device, int function, int register_num);
    void write_16 (int bus, int device, int function, int register_num, uint16_t value);
    uint8_t read_8 (int bus, int device, int function, int register_num);
```

void write_8 (int bus, int device, int function, int register_num, uint8_t value);
};

read_n

SYNOPSIS

```
#include <ipci_confspacerw_io.idl.h>

 uint32_t read_32 (int bus, int device, int function, int register_num);
 uint16_t read_16 (int bus, int device, int function, int register_num);
 uint8_t read_8 (int bus, int device, int function, int register_num);

 void write_32 (int bus, int device, int function,
               int register_num, uint32_t value);
 void write_16 (int bus, int device, int function,
               int register_num, uint16_t value);
 void write_8 (int bus, int device, int function,
              int register_num, uint8_t value);
```

DESCRIPTION

The `read_n` methods read an n bit register from the PCI configuration space. The `write_n` functions write the given value to the specified n bit register. This interface provides direct access to the configuration registers. Devices are not expected to use this interface, rather they should use the safe *IPCI_Conf* interface.

RETURN VALUE

On read the contents of the specified registers are returned. If the register does exist -1 is returned. Note that -1 may be a valid value for the contents of the register. It is assumed the caller knows if the register exists. The write function do not return any value.

A.3 Register space interfaces

A.3.1 IPCIMemSpace

#include "stdpax.idlh"
#include "mungi.idlh"

[
iid(0x79)
]
interface IPCI_MemSpace {
bus_space_t space_map (bus_addr_t address, size_t size, int flags);
};

space_map

SYNOPSIS

```
#include <ipci_memspace_io.idl.h>
#include <bus.h>

bus_space_t space_map (bus_addr_t address, size_t size, int flags);
```

DESCRIPTION

This method is used to obtain access to registers in the PCI Memory address space. `address` indicates the address of the registers in the PCI memory space, and `size` indicates the number of bytes to map. Currently no `flags` have been defined.

RETURN VALUE

On success a valid bus handle will be returned. On failure the `address` field of the handle with be NULL capability. Failure is possible if the requested space has already been mapped or is outside a valid range.

A.3.2 IPCIIOSpace

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid(0x92)
]
interface IPCI_IOSpace {
    bus_space_t space_map (bus_addr_t address, size_t size, int flags);
};
```

space_map

SYNOPSIS

```
#include <ipci_iospace_io.idl.h>
#include <bus.h>

bus_space_t space_map (bus_addr_t address, size_t size, int flags);
```

DESCRIPTION

This method is used to obtain access to registers in the PCI Memory address space. `address` indicates the address of the registers in the PCI IO space, and `size` indicates the number of bytes to map. Currently no `flags` have been defined.

RETURN VALUE

On success a valid bus handle will be returned. On failure the `address` field of the handle with be NULL capability. Failure is possible if the requested space has already been mapped or is outside a valid range.

A.3.3 IISAIOSpace

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x67 )
]
interface IISA_IOSpace {
    bus_space_t space_map (bus_addr_t address, size_t size, int flags);
};
```

space_map

SYNOPSIS

```
#include <iisa_iospace_io.idl.h>
#include <bus.h>

bus_space_t space_map (bus_addr_t address, size_t size, int flags);
```

DESCRIPTION

This method is used to obtain access to registers in the PCI Memory address space. `address` indicates the address of the registers in the PCI IO space, and `size` indicates the number of bytes to map. Currently no `flags` have been defined.

RETURN VALUE

On success a valid bus handle will be returned. On failure the `address` field of the handle with be NULL capability. Failure is possible if the requested space has already been mapped or is outside a valid range.

A.4 Interrupt interfaces

A.4.1 IPCI_Interrupts

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid(0x90)
]
interface IPCI_Interrupts {
```

```
    int register (cicap_t callback_instance, long interrupt);
};
```

register

SYNOPSIS

```
#include <ipci_interrupts_io.idl.h>

int register (cicap_t callback_instance, long interrupt);
```

DESCRIPTION

This function associates a PCI interrupt with a component instance. The component `interrupt_event` method is called when an interrupt occurs.

RETURN VALUE

0 is returned on success.

A.4.2 IISA_Interrupts

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid(0x91)
]
interface IISA_Interrupts {
    int register (cicap_t callback_instance, long interrupt);
};
```

register

SYNOPSIS

```
#include <iisa_interrupts_io.idl.h>

int register (cicap_t callback_instance, long interrupt);
```

DESCRIPTION

This function associates a ISA interrupt with a component instance. The component `interrupt_event` method is called when an interrupt occurs.

RETURN VALUE

0 is returned on success.

A.5 Generic device interfaces

A.5.1 IPCI_Device

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x666 )
]
interface IPCI_Device {
    void init (cicap_t pci_conf, bs_seq spaces, irq_seq irqs);
};
```

init

SYNOPSIS

```
#include <ipci_device_io.idl.h>

void init (cicap_t pci_conf, bs_seq spaces, irq_seq irqs);
```

DESCRIPTION

The `init` method is used to configure and setup a PCI device driver. `pci_conf` is a CICAP pointing to a `CPCIConf` instance, which the device driver can use to access its configuration registers. The `spaces` parameter contains a list of device mappings. The `irqs` parameter contains a list of the interrupt that have been registered to the device driver instance.

A.5.2 IISA_Device

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x999 )
]
interface IISA_Device {
    int init (bs_seq spaces, irq_seq irqs);
};
```

init

SYNOPSIS

```
#include <iisa_device_io.idl.h>

void init (bs_seq spaces, irq_seq irqs);
```

DESCRIPTION

The `init` method is used to configure and setup an ISA device driver. The `spaces` parameter contains a list of device mappings. The `irqs` parameter contains a list of the interrupt that have been registered to the device driver instance.

A.6 IO Completion interface

A.6.1 IIOComplete

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x270 )
]
interface IIOComplete {
    int complete (cicap_t from, uintptr_t key);
};
```

complete

SYNOPSIS

```
#include <icomplete_io.idl.h>

int complete (uintptr_t key);
```

DESCRIPTION

Called on the completion of an I/O operation.

RETURN VALUE Returns 0.

A.7 Stream interfaces

A.7.1 ISource

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x70 )
]
interface ISource {
    int set_sink (cicap_t sink);
};
```

set_sink

SYNOPSIS

```
#include <isource_io.idl.h>

int set_sink (cicap_t sink);
```

DESCRIPTION

The `set_sink` method is used to specify which component data should be sent to.

RETURN VALUE

Returns 0 on success.

A.7.2 ISink

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x71 )
]
interface ISink {
    int feed (cicap_t notify, buf_seq data);
};
```

feed

SYNOPSIS

```
#include <isink_io.idl.h>

int feed (cicap_t notify, buf_seq data);
```

DESCRIPTION

This command transfer the specified `data`, to the component. The `notify` parameter is the CICAP of the component that should be notified when the I/O is complete. This is usually the caller's CICAP.

RETURN VALUE

Returns 0 on success;

A.8 Packet interfaces

A.8.1 IPacketSource

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x147 )
]
interface IPacketSource {
    int register_packet_sink (cicap_t sink, int filter);
};
```

register_packet_sink

SYNOPSIS

```
#include <ipacketsource_io.idl.h>

int register_packet_sink (cicap_t sink, int filter);
```

DESCRIPTION

The `register_packet_sink` method is used to specify which component packets should be sent to. The filter argument is component specific and is used to specify a sub-set of packets.

RETURN VALUE

Returns 0 on success.

A.8.2 IPacketSink

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x71 )
]
interface ISink {
    int feed (cicap_t notify, buf_seq data);
};
```

10

feed

SYNOPSIS

```
#include <ipacketsink_io.idl.h>

int feed_packet (cicap_t notify, pkt_seq packets);
```

DESCRIPTION

This command transfer the specified sequence of `packets`, to the component. The `notify` parameter is the CICAP of the component that should be notified when the I/O is complete. This is usually the caller's CICAP.

RETURN VALUE Returns 0 on success;

A.9 Block interfaces

A.9.1 IBlock

```
#include "stdpax.idlh"
#include "mungi.idlh"

[
 iid( 0x271 )
]
interface IBlock {
        int write (cicap_t notify, block_seq data);
        int read (cicap_t notify, block_seq data);
        int get_block_count ();
        int get_block_size ();
};
```

write

SYNOPSIS

```
#include <iblock_io.idl.h>

int write (cicap_t notify, block_seq data);
```

DESCRIPTION

The `write` method writes a specified list of blocks to the block device. The `notify` parameter is the CICAP of the component that should be notified when the I/O is complete. This is usually the caller's CICAP.

RETURN VALUE

Returns 0 on success.

read

SYNOPSIS

```
#include <iblock_io.idl.h>

int read (cicap_t notify, block_seq data);
```

DESCRIPTION

The `read` method reads a specified list of blocks from the device into memory. The `notify` parameter is the CICAP of the component that should be notified when the I/O is complete. This is usually the caller's CICAP.

RETURN VALUE

Returns 0 on success.

get_block_count

SYNOPSIS

```
#include <iblock_io.idl.h>

int get_block_count ();
```

DESCRIPTION

The `get_block_count` method returns the number of blocks available on the block device.

get_block_size

SYNOPSIS

```
#include <iblock_io.idl.h>

int get_block_size ();
```

DESCRIPTION

The `get_block_size` method returns the size of the device's blocks in bytes.

Appendix B

Mungi Configuration Commands

B.1 Component related commands

B.1.1 load

SYNOPSIS

`load [-nonaming] [-nolocator] binary [component_name]`

DESCRIPTION

The `load` command is used for initialising components with the system. Components have an auto-generated main function which registers itself with the component system. The load command sets up an appropriate environment for execution of the main function. The main function returns a capability and component ID which are placed into relevant structures to allow other entities in the system to access the component.

The `binary` parameter indicates the location of the main entry point. This may be represented by a name in the Mungi naming service, or if the naming service is not yet registered, a DIT entry.

If a `component_name` the component's ID is placed into the naming service at the point.

The `-nonaming` flag indicates that the naming service has not yet been started and the loader should not try to use it.

The `-nolocator` flag indicates the the component locator has not yet been started.

B.2 Device initialisation commands

B.2.1 initcs

SYNOPSIS

`initcs component instance`

DESCRIPTION

The initcs, or init chipset, command creates the initial chipset device. The first argument identifies the location of the component. The second arguments is the name that should be given to the created instance.

B.2.2 isa

SYNOPSIS

```
isa component isabus [io=address:size] [irq=irq_num] instance
```

DESCRIPTION

The isa command is used to create an instance of an ISA device driver. The first argument is the name of the component. The second argument is the name of the ISA bus on which the device lies. A list of io parameters can be specified. Each one specifies the base address and size of a set of register in ISA I/O space to provide the driver with. A list of irq commands can also be specified. This registers the component to receive each of the listed IRQs. The last parameter is the name of the created instance.

B.2.3 pciscan

SYNOPSIS

```
pciscan pcibus
```

DESCRIPTION

The pciscan command will scan the provided PCI bus and attempt to load a device driver for each PCI device found.

B.2.4 enumdisk

SYNOPSIS

```
enumdisk diskcontroller
```

DESCRIPTION

The enumdisk commands interrogates the provided disk controller to determine how many block devices it supports. The command will retrieve all the available block device components and add them to the namespace at /dev/hd<n>, with incrementing values of n.

B.2.5 sink

SYNOPSIS

```
sink source_inst sink_inst
```

DESCRIPTION

The sink commands sets the source_inst component's sink to sink_inst. This is essentially a wrapper for the set_sink method.

B.2.6 packetsink

SYNOPSIS

`packetsink packetsource_inst packetsink_inst`

DESCRIPTION

The `packetsink` commands sets the `packetsource_inst` component's sink to `packetsink_inst`. This is essentially a wrapper for the `register_packet_sink` method.

B.3 Naming system commands

B.3.1 mkdir

SYNOPSIS

`mkdir dir_name`

DESCRIPTION

The `mkdir` command create a directory named `dir_name`.

B.3.2 ln

SYNOPSIS

`ln from to`

DESCRIPTION

The `ln` command create a hard link from `from` to `to`.

B.3.3 dit2file

SYNOPSIS

`dit2file dit_name real_name`

DESCRIPTION

The `dit2file` command copies the raw dit segments `dit_name` into a file called `real_name`

B.4 Miscellaneous commands

B.4.1 echo

SYNOPSIS

`echo [STRING]`

DESCRIPTION

The `echo` command simply prints a string to the console. This is useful for printing progress, and debugging, in the startup script.

B.5 Example

```
load -nonaming -nolocator clocator
load -nonaming naming /components/naming
echo Loading devices
load /dit/pciconf.app /components/pciconf
load /dit/pyxis.app /components/pyxis
load /dit/keyboard.app /components/keyboard
load /dit/serial.app /components/serial
load /dit/myshell.app /components/myshell
load /dit/ide.app /components/ide
load /dit/idehd.app /components/idehd
#load /dit/ether.app /components/ether
#load /dit/netstack.app /components/netstack
echo Making directories
mkdir /dev
mkdir /drivers
mkdir /drivers/pci
mkdir /drivers/pci/0x1095
mkdir /drivers/pci/0x1011
echo Linking components
ln /components/ide /drivers/pci/0x1095/0x0680
#ln /components/ether /drivers/pci/0x1011/0x0019
echo Initialising pyxis
initcs /components/pyxis /dev/pyxis
ln /dev/pyxis /dev/pci0
ln /dev/pyxis /dev/isa0
echo PCI bus scan
pciscan /dev/pci0
#ln /dev/pcidevice0 /dev/ether0
ln /dev/pcidevice0 /dev/ide0
ln /dev/pcidevice1 /dev/ide1
echo Enumerating disk
enumdisk /dev/ide0
enumdisk /dev/ide1
isa /components/keyboard /dev/isa0 io=0x60:1 io=0x64:1 irq=1 /dev/keyboard0
isa /components/serial /dev/isa0 io=0x2f8:8 irq=3 /dev/serial0
setbaud /dev/serial0 9600
#isa /components/serial /dev/isa0 io=0x3f8:8 irq=4 /dev/serial1
init /components/myshell /dev/shell0
init /components/myshell /dev/shell1
init /components/myshell /dev/shell2
init /components/myshell /dev/shell3
#init /components/netstack /dev/netstack0
sink /dev/shell0 /dev/serial0
sink /dev/serial0 /dev/shell0
#packetsink /dev/netstack0 /dev/ether0
#packetsink /dev/ether0 /dev/netstack0
dit2file string_data test
```

Appendix C

Machine specification

The machine used for testing and implementation was an Alpha Personal Workstation *au* series. The relevant details of the platform are outlined here:

- DC21164A (EV56) Alpha CPU running at 433 MHz.
- Data path width: 128-bits data plus 16-bits ECC.
- 384MB of ECC-protected Synchronous DRAM (SDRAM) main
- memory (SRAMs mounted on DIMMs).
- 64-bit PCI bus.
- ISA bus.
- 1 MB flashROM firmware.
- Enhanced IDE interface.
- Eight diagnostic LEDs.
- Mini-tower desktop enclosure.
- EIDE controller for CD-ROM drive.
- Ethernet: Fast Ethernet Lan controller for both 10- and 100-Mbps
- data rates.

Two Sil 0680 add-in IDE controllers were used during the testing. This controller support transfer speeds of up to 133MB/s, and supports the 48-bit addressing mode. Full details are available from the manufacturers website: http://www.siimage.com/products/sii0680.asp.

Two different IDE disk were used in the system. The first was a 160GB Maxtor DiamondMax D540X-4G driver. This drive supports transfer speeds of up to 133MB/s. The full details are available from the manufacturers website: http://www.maxtor.com/en/products/ata/desktop/diamondmax_d540x-4g/index.htm.

The other drive used was a 8.4GB Quantum Fireball. This drive supports transfer rates of up to 66.7MB/s. The manufacturer does not have online documentation for this drive.

Bibliography

[AH98] Alan Au and Gernot Heiser. *L4 User Manual*. School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, January 1998. UNSW-CSE-TR-9801. Latest version available from http://www.cse.unsw.edu.au/~disy/L4/.

[ASS95] Don Anderson, Joh Swindle, and Tom Shanley. *ISA System Architecture*. Mindshare, Inc., 1995.

[Bar96] Paul Ronald Barham. *Devices in a Multi-Service Operating System*. PhD thesis, University of Cambridge, 1996.

[Bau02] Andrew Baumann. A thread model for Mungi. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2002.

[Bla94] Richard John Black. *Explicit Network Scheduling*. PhD thesis, University of Cambridge, 1994.

[BMD99] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the 1999 USENIX Technical Conference*, pages 253–265, Monterey, CA, USA, June 1999.

[CH02] Kingsley Cheung and Gernot Heiser. A resource management framework for priority-based physical-memory allocation. In J Morris and F Lai, editors, *Proceedings of the 7th Asia-Pacific Computer Systems Architecture Conference (ACSAC)*, Monash University, Melbourne, Australia, January 2002.

[Cha01] Matthew Chapman. Low-latency user-level network drivers for Linux. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2001.

[Chu96] H. K. Jerry Chu. Zero-copy TCP in Solaris. In *Proceedings of the 1996 USENIX Technical Conference*, pages 253–264, 1996.

[Cla85] David D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th ACM Symposium on OS Principles (SOSP)*, pages 171–180. ACM Press, 1985.

[Com98] Compaq Computer Corp. *Alpha Architecture Handbook Version 4*, 1998. http://www.support.compaq.com/alpha-tools/documentation/current/alpha-archt/.

[Com99] Compaq. *Alpha 21264 Microprocessor Data Sheet*, 1999.

[COR99] *Interface Definition Language*, 1999. ISO/IEC 14750:1999.

[CV65] F.J. Corbato and V.A. Vyssotsky. Introduction and overview of the Multics system. In *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 195–195, 1965.

[CYC+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empiracal study of operating systems errors. In *Proceedings of the 18th ACM Symposium on OS Principles (SOSP)*, October 2001.

[DEC97] Digital Equipment Corporation. *DIGITAL Semiconductor 21174 Core Logic Chip Technical Reference Manual*, 1997.

[Dem97] Chris Demetriou. *Kernel Manual 1.5Y, Section 9, bus_space*. NetBSD, 1997.

[DP93] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on OS Principles (SOSP)*, pages 189–202, 1993.

[Edw00] Antony Edwards. A component architecture for system extensibility. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2000. Available from http://www.cse.unsw.edu.au/~disy/papers/.

[EH01] Antony Edwards and Gernot Heiser. Components + Security = OS Extensibility. In *Proceedings of the 6th Australasian Computer Systems Architecture Conference (ACSAC)*, pages 27–34, Gold Coast, Australia, January 2001. IEEE CS Press.

[EHL97] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. *L4 Reference Manual: MIPS R4x00*. School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, December 1997. UNSW-CSE-TR-9709. Latest version available from http://www.cse.unsw.edu.au/~disy/L4/.

[EKB+92] Joseph R. Eykholt, Steve R. Kleiman, Steve Barton, Roger Faulkner, Anil Shivalingiah, Mark Smith, Dan Stein, Jim Voll, Mary Weeks, and Dock Williams. Beyond multiprocessing—multithreading the SunOS kernel. In *Proceedings of the 1992 Summer USENIX Technical Conference*, pages 11–18, San Antontio, TX, USA, 1992.

[ERH96] Kevin Elphinstone, Stephen Russell, and Gernot Heiser. Supporting persistent object systems in a single address space. In *Proceedings of the 7th International Workshop on Persistent Object Systems (POS)*, Cape May NJ (USA), 1996.

[FBB+97] Bryan Ford, Godmar Back, Greg Benson, Jay Leprau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on OS Principles (SOSP)*, pages 38–51, St Malo, France, October 1997.

[FO71] R. J. Feiertag and E. I. Organick. The Multics input/output system. In *Proceedings of the 3rd ACM Symposium on OS Principles (SOSP)*, pages 35–41, 1971.

[GD96] Shantanu Goel and Dan Duchamp. Linux device driver emulation in Mach. In *Proceedings of the 1996 USENIX Technical Conference*, pages 65–74, San Diego, CA, USA, January 1996.

[GJP+00] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The Sawmill multiserver approach. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 109–114. ACM Press, 2000.

[Gra01] Charles Gray. Object naming in mungi. BSc (Hons) thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2001.

[Hei91] Joe Heinrich. *MIPS R4000 Microprocessor User's Manual*. MIPS Computer Systems, Inc., Sunnyvale, CA, USA, 1991.

[HLR98] Gernot Heiser, Fondy Lam, and Stephen Russell. Resource management in the Mungi single-address-space operating system. In *Proceedings of the 21st Australasian Computer Science Conference (ACSC)*, pages 417–428, Perth, Australia, February 1998. Springer-Verlag. Also available as UNSW-CSE-TR-9705 from http://www.cse.unsw.edu.au/school/research/tr.html.

[Hos94] Brad Hosler. Programming interface for bus master IDE controller, 1994.

[Int00a] Intel Corp. *IA-64 Architecture Software Developer's Manual*, February 2000. URL http://developer.intel.com/design/ia-64/index.htm.

[Int00b] Intel Corp. *IA-64 Architecture Software Developer's Manual Volume 2: IA-64 System Architecture*, January 2000. URL http://developer.intel.com/design/ia-64/index.htm, order no 245318-001.

[Int02] Intel Corp. *IA-32 Architecture Software Developer's Manual Volume 1: Basic Architecture*, 2002. URL http://developer.intel.com/design/pentium4/manuals/245470.htm.

[Kis01] Andrew Kisliakov. A Mungi framework for Linux device drivers. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, June 2001.

[L4K01] L4Ka Team. *L4* eX perimental *Kernel Reference Manual*. University of Karlsruhe, version 4-x.2 edition, October 2001. http://l4ka.org/projects/version4/l4-x2.pdf.

[Les02] Ben Leslie. User interrupt module for Linux, 2002. http://www.cse.unsw.edu.au/~benjl/user_interrupt.php.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles (SOSP)*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[McL02a] Peter T. McLean, editor. *Information Technology – AT Attachment with Packet Interface - 7 Volume 1 (ATA/ATAPI-7 V1)*. T13 Technical Committe, August 2002.

[McL02b] Peter T. McLean, editor. *Information Technology – AT Attachment with Packet Interface - 7 Volume 2 (ATA/ATAPI-7 V2)*. T13 Technical Committe, August 2002.

[MRC+00] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An IDL for hardware programming. In *Proceedings of the 4st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[Mun02] School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia. *The Mungi Kernel API, Version 1.2*, August 2002. http://mungi.org/src/.

[Nat95] National Semiconductors. *PC16550D Universal Asynchronous Receiver/Transmitter with FIFOs*, 1995.

[NBH02] Brett Nash, Ewa Z. Bem, and Gernot Heiser. Orthogonal persistence in the Mungi operating system. Submitted to the 26th Australasian Computer Science Conference (ACSC), September 2002.

[PCI94] *PCI IDE Controller Specification*, 1994.

[Pot99] Daniel Potts. L4 on uni- and multiprocessor Alpha. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 1999.

[PPTT91] Dave Presotto, Rob Pike, Ken Thompson, and Howard Trickey. Plan 9, a distributed system. In *EurOpen Conference*, pages 43–50, Tromsø, Norway, May 1991.

[PSW01] Christian Poellabauer, Arsten Schwan, and Richard West. Lightweight kernel/user communication for real-time and multimedia applications. In *11th International workshop on on Network and Operating Systems support for digital audio and video*, pages 145–154. ACM Press, 2001.

[R4695] Integrated Device Technology. *IDT79R4600 and IDT79R4700 RISC Processor Hardware User's Manual*, April 1995.

[Rit84] Dennis M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.

[Rub98] Alessandro Rubini. *Linux Device Drivers*. O'Reilly & Associates, Inc, 1998.

[SA99] Tom Shanley and Don Anderson. *PCI System Architecture*. Mindshare, Inc., 1999.

[Sam97] Samsung Electronics. *21164 Alpha Microprocessor Hardware Reference Manual*, 1997.

[SMLE02] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th ACM SIGOPS European Workshop*, September 2002.

[Ste01] Shane Stephens. Framework for biomedical applications on PLEB. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2001.

[Ste02] Shane Stephens. BiOS - a biomedical development platform, 2002. http://www.cse.unsw.edu.au/~bios/.

[TR74] Ken Thompson and Dennis M. Ritchie. The UNIX time-sharing system. *Communications of the ACM*, 17:365–375, 1974.

[VERH96] Jerry Vochteloo, Kevin Elphinstone, Stephen Russell, and Gernot Heiser. Protection domain extensions in Mungi. In *Proceedings of the 5th IEEE International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 161–165, Seattle, WA, USA, October 1996.

[VRH93] Jerry Vochteloo, Stephen Russell, and Gernot Heiser. Capability-based protection in the Mungi operating system. In *Proceedings of the 3rd IEEE International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 108–15, Asheville, NC, USA, December 1993.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned scalable internet services. In *Proceedings of the 18th ACM Symposium on OS Principles (SOSP)*, pages 232–246, October 2001.