TECHNISCHE UNIVERSITÄT
CHEMNITZ

Faculty for Computer Science
Operating Systems Group

Master Thesis

Summer semester 2018

PylotOS - an interpreted Operating System

Submitted by: Naumann, Stefan
stefan.naumann@informatik.tu-chemnitz.de

Programme of study: Master Informatik
3rd semester
Matriculation number: XXXXXX

Supervisors: Christine Jakobs
Dr. Peter Tröger
Prof. Dr. Matthias Werner

Submission deadline: 11th July 2018

Contents

List of Figures

Listings

List of Acronyms

ACL Access Control List
ACPI Advanced Configuration and Power Interface
APC Asynchronous Procedure Call
API Application Programming Interface
APIC Advanced Programmable Interrupt Controller

BIOS Basic Input Output System
BSB Back Side Bus

CIL Common Intermediate Language
CPU Central Processing Unit

DCB Domain Control Block
DMA Direct Memory Access
DPC Deferred Procedure Call

ELF Executable and Linking Format

FSB Front Side Bus

GC Garbage Collection
GID Group Identificator
GIL Global Interpreter Lock
GPIO General Purpose Input Output
GUID Globally Unique Identifier

HAL Hardware Abstraction Layer

I/O Input/Output
IDL Interface Description Language
IDT Interrupt Descriptor Table
IPC Interprocess Communication
IRP I/O Request Package
IRQ Interrupt Request
IRQL Interrupt Request Level
ISR Interrupt Service Routine

JDK Java Development Kit
JIT Just-In-Time
JVM Java Virtual Machine

MDL Memory Descriptor List
MMU Memory Management Unit

pc Program Counter / Instruction Pointer
PCB Process Control Block
PCI Peripheral Component Interconnect
PIC Programmable Interrupt Controller
PID Process Identificator
PnP Plug and Play

RAM Random Access Memory
RMI Remote Method Invocation
RNG Random Numbers Generator
RPC Remote Procedure Call

SATA Serial ATA, Serial AT Attachment
SIP Software Isolated Process
sp Stack Pointer
syscall System Call

TCB Thread Control Block
TLB Translation Lookaside Buffer

UART Universal Asynchronous Receiver Transmitter
UID User Identificator
USB Universal Serial Bus

VM Virtual Machine

WDM Windows Driver Model
WMI Windows Management Instrumentation

1. Introduction

Nearly every hardware component of computers is becoming more powerful and cheaper. The clock-rate and parallelism of processors is increasing. With better internal design, memory access time decreases while memory capacity grows. Disk access times decrease and the capacity is increasing. There are many methods for controlling the increased complexity of hardware, for example by using design patterns or object orientation. But system software is still complex, hard to understand and maintain. Most operating systems are still written in low-level-languages like C.

An operating system, written in Python may be able to change that, for it may use object-orientation and higher level constructs, therefore making it easier to read and maintain the operating system code. On the other hand, using higher level constructs may result in less programming bugs. There already is operating system software, which is object-oriented, but nearly always in a compiled-language like C++.

Python, or script-languages in general, are executed using an interpreter, a run-time-software, which reads the program code and translates it to machine-instructions on run-time. This concept should not be confused with running potentially interpreted code in a compiled manner, i.e. compiling code of a potentially interpreted language before executing the program. Having this layer of abstraction of the real hardware brings new problems. Some concepts or functionalities may not exist in the script-language, for example direct register access. Either the interpreter needs to be changed, therefore using a non-standard version of the script language. Otherwise these functionalities may be implemented as function-call in the interpreter or the language-library it contains.

Tröger et al [23] identify three models for execution of operating systems it calls compiled first level, interpreted second-level and interpreted first level operating system. These models are depicted in Figure 1.1. Most classic system software can be considered compiled first level operating systems, where the kernel is written in a compiled language and run as native machine code on the processor directly, without the need for an interpreting environment. User applications can be run on top of that compiled kernel, including an interpreter.

The interpreted second level operating system model uses a compiled first level OS. On that an interpreter executes the interpreted second level OS. These mostly use the facilities of the underlying operating system to provide functionalities like memory management or drivers. Although the first level kernel is minimal in these builds, many functionalities used by the interpreter itself are still implemented in this kernel.

An interpreted first level operating system uses the interpreter as kernel and has just enough glue code to start executing the script-language. The goal is to have as much functionality as possible in the level of the interpreted language, and as little as possible in a compiled language including the interpreter. This thesis will focus on the architecture of an interpreted first level operating system.

A classic operating system takes care of many tasks, not the least of which it constructs an abstraction between the hardware and the software. This includes representing and managing hardware resources for the use of several user programs or so called processes. Most classic operating systems

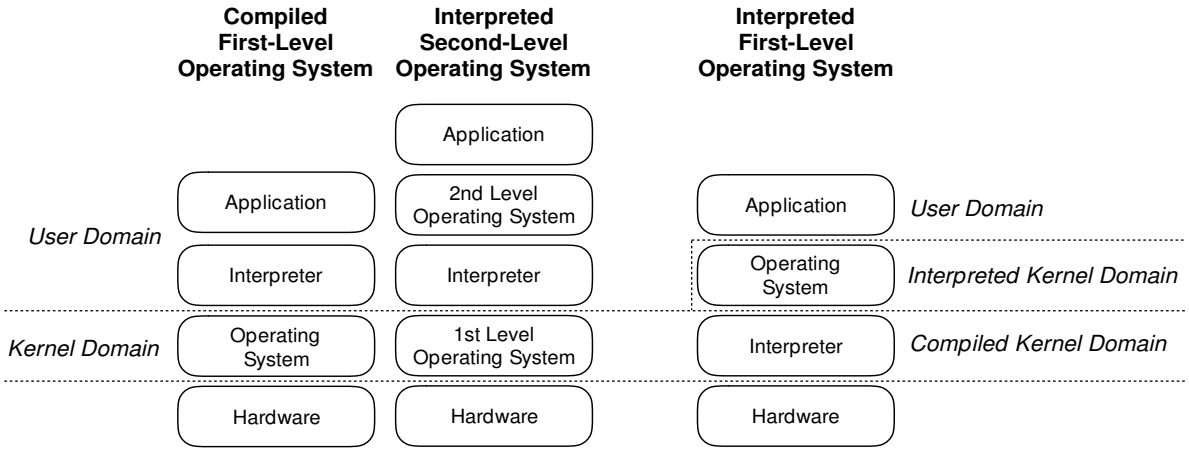

Figure 1.1.: Execution models of operating systems [23]

include the facilities for processes to be isolated from each other, i.e. protected against faulty or evil spirited processes, while also ensuring they can communicate via the means of an Interprocess Communication (IPC) functionality. Managing hardware includes initialising the devices and using them when needed by a process, ideally representing them in a generic fashion, i.e. it requires an architecture for drivers to be integrated into the operating system.

This thesis describes the architecture of an interpreted first level operating system developed in the Python programming language. This OS should be as generic as possible, i.e. it should be as capable as a conventional OS. It should also support as much dynamics as a conventional OS. Making the system more static would solve most of the problems commonly found in operating systems, and therefore make the problem trivial.

The target operating system may however not be restricted to only running user processes, which are written in Python, but also ones compiled into the kernel or loaded as programs on run-time. These processes would need some sort of interoperability layer to be able to communicate with the interpreted processes when needed. Also additional security and isolation facilities need to be implemented to protect interpreted processes, the interpreter and the interpreted operating system kernel from modifications by these compiled processes.

After discussing some preliminary considerations like the basic terminology, a textbook operating system architecture and the architecture of the x86 and ARMv6 platform, there will be a deeper look into related work in the field. Then existing libraries and interfaces will be considered, as well as conventional and unconventional operating system's architectures. From that a suitable architecture for the interpreted first level operating system will be derived.

The proposed architecture is outlined with code listings, but this code is not to be seen as finished or tested, but as a rough draft of what the architecture may look like in Python. A prototype may use the Micropython Python interpreter which is optimised for embedded ARM hardware. This thesis will use the Python programming language as target language.

2. Preliminary Considerations

This chapter will describe the textbook approach to operating systems [22]. Then it will explain the general hardware for x86-based systems [3,21] and the Raspberry Pi as target platform for PylotOS, as it is key to know the target platform when designing an operating system.

2.1. Textbook Operating System Architecture

Designing a new operating system needs knowledge about the basic theory of operating systems alongside knowledge of the target platform and reference operating systems. Especially developing the concept for a first level interpreted operating system requires deep understanding of the features and tasks of an operating system.

An operating systems carries out two tasks [22]:

1. provide the user applications with a clearer, better, and simpler model of the computer
2. manage resources

A user application, which is designed and developed to run on top of the OS, is seen as the customer of the OS, not the developer. A user application is based on certain interfaces and call semantics being present in the operating system. It needs these for running.

Managing resources is the other main task of an operating system. There are loads of resources in a computer, from the hardware devices, like the processor, main memory to the bus adaptors, interrupt controllers up to graphics cards, keyboards, mice, etc. Besides the physically present devices, the system can create virtual resources like files, or messages to make using the devices simpler (falling into task 1).

Generally from the top-down perspective is that the OS defines and implements interfaces, and user applications use these abstractions to solve problems. Viewing the system bottom-up yields to the OS ensuring orderly and controlled allocation of devices, the CPU, and memory among the programs, which require them.

Tanenbaum [22] defines an operating system as the software on a computer, which runs in supervisor mode of the CPU, the rest runs in user mode and is therefore seen as user applications. The OS runs on the bare hardware and provides the base for other software to run. Also the user applications can be exchanged at run-time, whereas the OS cannot. The distinction between user and supervisor mode can be blurred on embedded systems, where no mode change may exist or interpreted environments like Java.

A user application can request functionality using system calls (syscall). These are different from calls of local functions in many operating systems, as they require the CPU mode to switch from user mode to kernel mode and may require also the usage of a different stack among other changes. In most operating systems, user applications are prohibited to access operating system data structures and only the operating system can access these structures, when the CPU is returned to supervisor mode.

2.1.1. Processes and Interprocess Communication

A process can be seen as a program in execution. A process has an own address space, which contains its code, data and execution stack. An execution stack is a piece of memory, where the calling history is stored alongside the local variables of the currently running subroutine. The process also has its own list of opened files, CPU registers, out-standing alarms and list of related processes in most operating systems. A process can in a way be interpreted as a container for resources.

The operating system can create something called pseudo-parallelism with processes, by interleaving their execution, i.e. executing one process for tens of milliseconds, then switching to another process and so on. This method stays in contrast to true hardware parallelism for example in a system with several processors.

Every software can be seen as a sequential process (short: process). Conceptually every process is run on its own processor, the interleaved execution of several processes is not visible to the processes. For them it looks like they are run on a virtual processor, be it on a slower one than the physical processor.

Creation of processes is done by a process, which uses a process-creation-syscall (system call). This system call instructs the operating system to create a new process and indicates, what program to run in it.

UNIX-systems use `fork` and `execve` to create an exact copy of the calling process and load another memory image. Windows uses the `CreateProcess`-API call to create a new process and load a new program in it. Parent and child process have their own independent address space.

Termination of processes is done by calling the `exit` syscall in UNIX or `ExitProcess` in Windows respectively. This tells the operating system, that the process has done its work and can be terminated. Another process on UNIX may `kill` another process.

A Hierarchy of processes does not exist in all operating systems. In UNIX the calling process becomes the parent of the child process, which creates a tree-like structure. It also dictates, which process group a process is member of. In Windows such a hierarchy is not present, but the parent gets a handle to the new process to manipulate it. But the parent may choose to transfer the handle to another process.

Process States are present because processes may need to interact or wait on input. When a process blocks because it is waiting and logically cannot continue it is blocked. It can be unblocked, when an event occurs, which unblocks it. For example a process may wait for user input and is blocked, because the input is not present at the moment. When the user inputs something the process is unblocked, and returned to the *ready*-state.

The processes in the *ready*-state can be scheduled and executed by the processor. The currently running process (single processor system) or processes (multiprocessor system) are in the *running*-state. Figure 2.1 depicts the states and the transition between them.

Implementation A process table exists in the operating system, where each entry contains information about a process, sometimes called Process Control Block (PCB). These Process Control Blocks (PCBs) contain information for process management and context switching, i.e. switching from one process to another one, memory management and file management. These information include the CPU registers, program counter, status words, stack pointer, process ID, accounting information, like

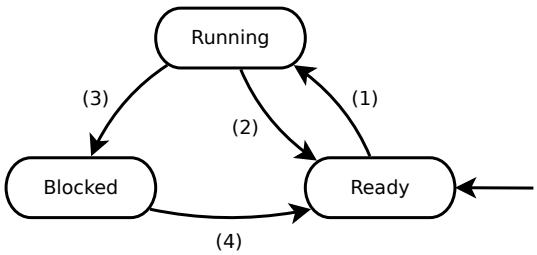

Figure 2.1.: Process states and transitions [22, pg. 92]
(1) A process is scheduled and therefore transitions to the *running*-state.
(2) The running process is pre-empted or yield the CPU, so transitions from *running* to *ready*.
(3) A process is blocked, i.e. it waits for input or data, which is not ready at the moment. It cannot be scheduled until unblocked.
(4) The data, the process waited for, became ready, so the process is unblocked and can be scheduled.

CPU time used, user IDs, group IDs, file descriptors for opened files, the working directory, etc.

When a process is pre-empted, because an Interrupt Request (IRQ) occurs the state of the process has to be saved, so it can be restored, when the handler function is finished. The following steps have to be done so an interrupt handler function can return to the pre-empted process correctly [22, pg. 96]:

1. Hardware saves program counter on the stack
2. Hardware loads new program counter from an interrupt vector
3. Assembly language procedure saves general purpose registers
4. Assembly language procedure sets up a new stack (or switches to already present kernel stack)
5. C interrupt handler runs and services the interrupting devices (typically buffers input)
6. Scheduler decides what process to run next
7. Assembly language procedure starts up the new current process
   a) Switch to the process's stack
   b) Load its general purpose registers
   c) Load the program counter from its stack (i.e. returning to the process)

Threads

Threads allow parallelism inside the container of a process. Several threads can exist in one process, all sharing the same address space, opened files, and other resources. A thread is more lightweight than a process. When a thread is blocked, another may take over and waiting times can thereby be overlapped with the execution of other code in the same context.

A process is a container for resources, like the address space (text, data) or opened files, whereas threads are the units of execution (which have their own program counter, registers, stacks and thread state). Threads are the scheduling entities. Until there we implicitly assumed, that there is only one thread of execution inside a process.

Threads are however not as independent of each other as processes. A thread can access and modify every address in the address space including stacks of other threads. Protection between threads does

not exist, because it is impossible and should not be necessary. Processes are assumed to be written by several developers and to be hostile to each other, whereas threads of one process are assumed to be owned by one user and are created to cooperate and not fight.

A process starts with a single thread, which can create new threads (parameter: procedure to start the thread on). Threads can also exit, can be waited for or give up the CPU to let another thread run. The IEEE standard 1003.1c describes POSIX threads (pthread), which is an implementation of threads.

Threads can be implemented in user space or kernel space, each with its own advantages and disadvantages.

Interprocess Communication

Sometimes a process may want to communicate with another process. There are several problems, like:

- How is the data transmitted from process A to process B?
- Processes should not get in each other's way.
- Sequencing is a problem (B needs input from A, has to wait until A produces output).

The same problems, except the first one, occur also when working with threads. Transmitting data is trivial, as threads can use the shared address space. Processes have no shared address space, so the operating system has to be invoked to transfer data from process A to B.

When several processes or threads share resources, the part of the programs are called critical sections. It depends on the order of the execution of critical sections (and their interleaving) which result is produced. In the worst case, the shared resource may be corrupted. This situation is called race condition. For resolving that mutual exclusion has to be ensured. When a process accesses a shared resource, no other process is allowed to access it at the same time.

There are several methods for securing mutual exclusion, like Semaphores, Test and Set-Instructions or monitor-objects. In the context of this thesis these not relevant and are therefore not described here.

2.1.2. Driver Model

One task of an operating system is the abstraction of an ugly interface (in this case hardware) to a more user-friendly interface. A driver is a program, which controls some peripheral hardware.

Hardware

Input/Output (I/O)-devices may be divided into two classes: block devices and character devices. Block devices use information in fixed size blocks, which can be addressed in blocks. Every block may be read or written independent from any other block at any time. Character devices use data streams instead of blocks, so they deliver a data stream or they accept a data stream, without respect to any block-organisation of the data. There is no addressing and searching-operation.

This classification is not perfect, there are devices which do not fit into any of these classes. Clocks have no block-oriented data, nor data streams, they only trigger interrupts after a certain time.

Addressing the I/O-devices

The device controller has several registers for receiving and presenting data to the CPU. There are two different ways of addressing these registers: memory-mapped or port-mapped.

Memory-mapped I/O states, that the registers of the controller are mapped into the same address space of the memory. So the CPU cannot distinguish between an address, which leads to a memory-cell in main memory and an address, which points to a register of a hardware controller. There has to be additional hardware, which translates an address to the correct target – either the correct hardware controller or the memory-cell. Also the caching-strategy used by the CPU may normally prevent reading current values from the device, so it has to be disabled for all addresses which do not lead to memory.

A different method is port-mapped I/O, which associates a port (number) to a device. Then the device can be used with different commands, examples instructions like `in` and `out`. With every I/O-command used, the programmer must specify the port-number for the device, which the program wants to use.

A downside of the port-mapped method is, that it uses special commands, which are not mapped into languages like C or C++. When using memory-mapped I/O the addresses can be used in C regardless of which device the address really points to. The OS may want to enforce a protection mechanism, where it prevents certain processes from accessing certain devices. So it may need to set the processor into a different state disallowing port-mapped I/O. With memory-mapped I/O it may just use virtual addresses and a translation unit, which is built into most CPUs these days.

Interrupts and exceptions

Hardware runs concurrently to the CPU and therefore there can be hardware events happening when the CPU is busy with executing some other code. There are two basic methods of collecting data from a device: polling and interrupt-based. With polling the operating system reads the registers of the controller after a certain amount of time. According to their values the operating system determines, whether an event has occurred, and decides what to do.

The other method is interrupt-based event handling. The CPU has interrupt-lines, which signal the CPU, whether there are events, that need to be taken care of. The processors instruction cycle contains not only the stages for fetching, decoding and executing the instruction itself, but also checking the interrupt lines. If one is set it may save some data of the currently running code (depending on the processors architecture), and jump into the appropriate Interrupt Service Routine (ISR). The ISR stores the rest of the context. It then decides from the set interrupt lines, the state of the interrupt controller and register-values of the devices which device raised the interrupt and tries to resolve the issue. Afterwards the CPU will return to the process it pre-empted.

According to the state of the processor the operating system needs to do more, so that the interrupted process can be restored. Also, if virtual memory is used, the paging-tables may be switched, from the process, to the kernel address-space. Interrupt / exception handling is a very specific topic, which heavily depends on the architecture used.

A different source, which can interrupt the CPU is the CPU itself. Exceptions, like Divide-by-Zero or a memory access to an unmapped address, are situations, where the processor itself decides to interrupt the current flow of the program, because of an erroneous state, which it cannot resolve itself. An exception also triggers an ISR or sometimes called exception handler, which identifies the fault that

occurred and tries to resolve it. This can mean for example to swap a memory page back into memory, or to terminate the running program, because it cannot be continued.

Exceptions have many similarities with ISRs, but they technically are not considered part of the I/O-subsystem. In this thesis they will be considered only partly, as the main focus lies on I/O. However exceptions are the sole method of communication between the user programs and the operating system's services, therefore the solely method of invoking the functionalities of a driver.

Copying blocks from a block device

Programming data-transfer to or from a device may be done in several different ways. A driver wants to communicate with a device, for example a line-printer, which prints the characters as it receives them. When it receives a letter, a value in the status-register is set to `not ready`. Then it prints the letter and resets the value to `ready`. Characters which are received when the device is not ready will not be printed. One way of implementing the printing-function may be, to send the first character, then busy wait, until the printer is ready to receive the next character. Then send the next character. This might be a valid approach when the waiting times are very short. When they are longer it might be better to execute a user process until the printer gets ready again.

The printer may trigger an interrupt, whenever it becomes ready again. The print-routine may be changed to sending a character, then waiting for a signal to be raised, before sending the next character. When the printer triggers an interrupt the ISR raises the signal for the print-routine and, when it is executed the next time, it will send the next character and wait again for the signal. This approach may be a bit more efficient, but it involves many interrupts to be sent and received, so the system may be busy with saving the status of the running processes, raising the signals for the print-routine and returning from the ISRs.

Better is the use of a Direct Memory Access (DMA)-controller to copy a block of data from the main memory to the device. The DMA knows how to send the data to the I/O-device. The operating system sets the registers of the DMA-controller, filling in the size of the block and a pointer to the block itself. The DMA then reads the memory-block and outputs it to the I/O-device. When finished, the DMA triggers an interrupt to the CPU. The ISR raises a signal, which in turn unblocks the print-routine, which then sends more data to the printer or returns.

Structure of drivers

Tanenbaum [22] proposes a layered structure of driver-code in the operating system. The bottom-layer is the hardware itself, which can trigger interrupts. These interrupts are caught by the appropriate ISRs on the next layer. These can store data or execute the driver-software itself, which handles the interrupt. It can also save a notification into a queue, so the driver can handle the situation later on. The ISRs may need to send some data to the user, this has to be done through the device-independent operating system software, which then hands over the data to the user-software. The user-software does not have to care about the interface of the driver itself, it only uses the device-independent operating system code, which creates an abstraction from the driver-software.

The device-independent code helps creating a uniform interface for all drivers and the user. The driver has to provide a specific interface, otherwise it will not work with the device-independent software. Specifying this interface makes creating an operating system easier, so that operating system developers do not have to care about every driver and their interfaces, but need to form a generic driver interface, that works with drivers for most to all devices.

Another important task of the device-independent code is the coordination of device-accesses. There are devices, which may only be used by one process at a time, for example a printer. Implementing this functionality is done by creating a spooler, the only process, which has permissions to access the printer. Other processes have to send the data, they want to print, to this spooler, where the data is queued. The spooler decides, what request will be printed next, when a request is done.

The device-dependent code may be allowed to call some kernel-functions, but this is restricted to some specific ones. Drivers need to support re-entrance, so they need to take into account, that they might be pre-empted because of an interrupt or even called a second time, for a different device of the same kind.

So there are two main ways drivers are used: They can be called be the user-processes and they can be called for handling an interrupt or a queued notification. For both of those usages there has to be a structured way for invoking the driver's functionalities.

2.2. Platform Considerations

To understand the later descriptions of the Windows and Linux driver models the x86 platform has to be well-known. The next section will describe the x86 platform and its way of handling Interrupt Requests (IRQs) and I/O operations. Then the Raspberry Pi platform will be depicted with its way of handling peripheral devices and the similarities and differences to the x86 platform.

2.2.1. Intel x86

An x86 system incorporates a system bus (for example Peripheral Component Interconnect (PCI)), which connects most of the devices in the system. A bus is a data path between devices, and therefore the primary communication channel inside a PC. Other buses can be connected by so called bridges to the system bus or other buses.

There are two high speed buses from the CPU, the Front Side Bus (FSB) to the Random Access Memory (RAM) and the Back Side Bus (BSB) to the external hardware cache. The host bridge connects the system bus to the FSB. Every device in a PC is hosted by exactly one bus. The bus affects how the kernel has to treat the device. An I/O bus connects devices to the CPU by the means of I/O ports, I/O controllers and I/O devices.

An x86 processor can access so called I/O ports, i.e. I/O addresses with the instructions `in`, `ins`, `out` and `outs`. An I/O port is an eight bit register of a hardware device (normally outside of the main CPU), which can be written to or read from. x86 uses 16 bit addresses to access I/O ports, meaning an x86 system can have a maximum of $2^{16} = 65536$ I/O ports. It is possible to transfer more than 8 bit at a time, by concatenating more than one port, but their address has to be aligned.

I/O ports can also be mapped into the physical memory space, so they can be accessed by `mov` instructions. `in` and `out` are seen as a slower way of accessing the hardware, whereas memory mapped I/O ports are seen as a more efficient mean, as they can be used by DMA.

An I/O interface is a hardware circuit that connects the device controller with the I/O ports. It acts as an interpreter of the values of the device registers (I/O ports) and sends commands and data to the controller based on their values. It also detects changes in the device to update I/O ports and the status register(s). The I/O interface may also use an interrupt line to the Programmable Interrupt

Controller (PIC) to issue an interrupt to the CPU.

An I/O controller operates the device based on commands. It converts high level commands to sequences of electrical signals and vice versa. Not all devices need dedicated device controllers.

Interrupt handling

Intel x86 distinguishes between exceptions, which are synchronous traps issued from the CPU itself when an error occurs (like a page fault), and interrupts, which are asynchronous traps issued by external devices [3]. Exceptions and interrupts can be handled basically the same way, but this thesis will focus on interrupt handling. Some IRQs are maskable, i.e. they can be disabled at run-time by the CPU, whereas non-maskable IRQs cannot be disabled.

An Programmable Interrupt Controller (PIC) is an interrupt controller, which is placed 'externally' to save lines to the CPU. It basically acts like a multiplexer of interrupt lines. In x86 systems there are two frequently used controller, the PIC (two cascaded 8259A style chips) used in uniprocessor systems and the Advanced Programmable Interrupt Controller (APIC). This thesis will not go in detail about the interrupt controllers. Also the abbreviation PIC is used to mean both PIC and APIC.

A PIC detects raised interrupt signals by hardware devices. If several lines are raised decides based on a set of rules or a table which of these lines should be passed (first) to the CPU. After choosing one, it will store the interrupt vector of the line in an I/O port and raise the `INTR`-line of the processor, issuing an IRQ to the processor itself.

The processor jumps to an address listed in an Interrupt Descriptor Table (IDT), when its `INTR`-line is raised by the PIC. This table needs to be set-up before the kernel enables (local) interrupts, because if the interrupt line is already raised, but no (or the default) address is specified unwanted behaviour might occur.

The ISR will read the interrupt vector from the I/O port and decide which action has to be done based on the vector. When a device raises an interrupt line, it needs to be serviced and the interrupt needs to be acknowledged by the CPU. Therefore to acknowledge the interrupt the CPU writes to a specific I/O port of the device and the PIC after it has done the necessary work. The device will then lower its interrupt line.

Power handling

The Advanced Configuration and Power Interface (ACPI) contains six system states for power handling. Table 2.1 lists the six states.
The states S0 to S4 retain enough data on disk or in memory to move back to S0 and resume operation. ACPI also defines four states for devices D0 through D3, where D0 stands for the fully on state and D3 for fully off. The semantics of D1 and D2 are defined by the device drivers.

Direct Memory Access (DMA)

A bus master is a device, which can use the address and data bus and therefore control the contents of the memory. With PCI every peripheral can act as bus master, not only the CPU as before, so they can load or store data from or in the RAM on their own. This is mostly used with disk or mass storage devices, which need to transfer loads of data.

Table 2.1.: ACPI power states as described by [21]

State	Power Consumption	Software Resumption	Hardware Latency
S0 (fully on)	maximal		
S1 (sleeping)	< S0, >S2	where it left of → S0	< 2 seconds
S2 (sleeping)	< S1, > S3	where it left of → S0	≥ 2 seconds
S3 (sleeping)	< S2	where it left of → S0 processor is turned off	≥ 2 seconds
S4 (hibernating)	trickle current to power button and wake-up circuitry	system restarts from hibernating file from disk, resumes from there → S0	undefined long latency
S5 (fully off)	trickle current to power button	system boot	undefined long latency

A system might have auxiliary DMA circuits, which can be activated by the CPU to copy data from a device to memory or vice versa without further help of the CPU. When the transaction is finished an interrupt is issued. Conflicts between the CPU and a DMA circuit are resolved in hardware by a memory arbiter.

One can distinguish between asynchronous DMA and synchronous DMA [3]. An asynchronous DMA is triggered by a hardware device. For example a network interface card might receive a frame and issues an IRQ to the CPU. The ISR will acknowledge that IRQ and start a DMA transfer to copy that frame to memory. When the DMA transfer is finished, another interrupt is issued to allow a driver to react to that new frame.

A synchronous DMA is triggered by a process. For example a user application wants to play some sound samples through the sound card. The driver accumulates the samples in a kernel buffer and instructs the DMA circuitry to copy the buffer to the sound card. When completed, an interrupt is issued. The driver has then the option to start another DMA transfer if there are uncopied samples left.

For DMA the hardware has to have direct access to main memory and therefore must be able to address certain areas of the main memory. The main CPU however may use different addresses from the hardware devices and maybe even use a virtual memory management, where the address space is divided into pages, which reside on 'random' locations in the memory.

The Intel x86 platform uses a the physical addresses, which are issued by the CPU also as addresses for the hardware devices, that means, that the physical addresses can be used by hardware devices to do DMA. However other hardware architectures may not and may need to set-up a special I/O Memory Management Unit (MMU), before a DMA transfer can be started.

2.2.2. Raspberry Pi

The Raspberry Pi 1 and Zero features a BCM2835 chip, which includes the main processor core and many peripheral hardware device controllers. The main CPU is an ARMv6-family processor, the ARM1176jfz-s. The processor contains sixteen 32 bit general purpose registers. Some of them have another function, like the link register, stack register or program counter, which are also visible as normal general purpose register. The program status register (`cpsr`) cannot be accessed by `mov`-instructions but by `mcr` and `mrc`.

Booting the Raspberry Pi is a bit different from normal PCs, because the first thing to boot-up is the VideoCore IV graphics processing unit. It will open a certain file on the SD-card and execute it. Under normal circumstances, i.e. with standard Raspbian start-up code, it will then load the kernel, initialise hardware devices and start the main ARM CPU.

The registers of most hardware devices are mapped into the address space of the CPU [4]. There is one exception to that rule. The registers of the C15 co-processor cannot be accessed by load and store instructions to memory addresses, but rather have to be accessed by special instructions (`msr`, `mrs`). Functions of this processor include enabling the MMU, activating instruction and data caches, enabling branch prediction or allowing the CPU to load and store unaligned data word or half-words [2].

The BCM2835 contains an interrupt controller [4, pg. 109]. The main CPU only has seven trap handlers. Several of them handle exceptions, i.e. problems that occur inside the CPU on program execution like an access to an unmapped section or an undefined instruction. The interrupt table is set up on the address 0x00000000 (of the ARMs address space).

The C-routine would need to store the registers of the processor and put the link register on the stack for being able to restore the context of the pre-empted process. If the stack needs to be changed, the stack register also needs to be stored in a globally known variable. After the interrupt has been handled, the process' context needs to be restored and a return from interrupt instruction issued.

Interrupt Controller The Raspberry Pi uses an 'external' interrupt controller. It indicates, which device issued an interrupt. The ISR can query its registers and find clues, what devices issued an interrupt, and therefore which devices (or device drivers) needs checking next.

The interrupt controller allows masking and unmasking interrupt sources with the disable and enable-registers. The controller has 64 interrupt sources or lines, which can be triggered, but not all of them are used. The ARM1176jfz-s uses a bit in the program status register (`cpsr`) to mask (disable) interrupts and fast interrupts all together.

3. Existing Implementations

This chapter contains a look into some existing operating system kernels, their driver interfaces as well as certain other aspects of their design like process implementations. The first section however describes several operating systems very abstractly and chooses some, which will be examined in closer detail throughout this chapter. The next one is about the C library, which defines some facilities for user programs to use system functionalities and in turn driver functionalities or process management. Then the selected operating systems are examined closer.

3.1. Related Work

When developing a new operating system especially on a field like an interpreted operating system, it is wise to look at other operating systems and their design choices in the past. First there are conventional operating systems, which are nearly completely written in compiled languages like C or C++. They use a compiled kernel in whatever instruction set the processor supports. This thesis will look into some conventional operating systems like Linux [3], Windows [21], and a BSD-variant [13]. Operating systems like Android or Chromium OS count to this group, as they all use a compiled Linux kernel to host their applications. They are however not outlined closer. There are also approaches to implement an operating system in odd languages like Haskell [10].

Although Android [14] is not a first level interpreted operating system, it does show some interesting features. Android basically incorporates a Linux kernel, which runs most of the necessary administrative tasks. It handles scheduling and memory management. However one of the first things to start after booting, is the virtual Java-esce machine, formerly called Dalvik, now Android uses the Android Runtime (ART). Android does not use the standard Bytecode of Java, but an own one, which is tailored more towards being small and lightweight.

It will be spawned with an Zygothe process, which acts as an init-process of conventional operating systems like 4.3BSD (see Section 3.3.1). The Zygothe will load certain libraries, which will be shared by every other Java-esce process in the system. The Dalvik-processes are mapped directly to Linux-processes and are scheduled as such. Android is too close to Linux to be examined in more detail.

Besides the conventional operating systems, many groups researched interpreted operating systems or writing operating systems in a potentially interpreted language, but opted for Just-In-Time (JIT)-compilation for performance reasons. Some contenders include Singularity [12], Inferno [5], or JX [8].

Yet a different approach can be seen in the picoJava-I processor [18]. The processor executes Java bytecode as its native instruction set. A processor running Java byte code can't really be told about from a classic x86 or ARM processor. All of these processors execute some sort of instructions as native code. So an OS for the picoJava-I would be considered a first level compiled OS, therefore not what this thesis is trying to develop.

In other words: picoJava-I executes Java byte code which was created from Java source files. An x86 processor executes x86 code, created from (for example) C-source files. Both of these native codes can be interpreted in software, Java inside a regular Java Virtual Machine (JVM), x86 code with QEMU.

The interesting part for the first level interpreted operating system is not running as software natively in hardware, like Java byte code on picoJava-I, but rather running inside an interpreter software on any hardware platform.

The Singularity project [12] aims for a rethinking of operating system design. The operating system is written in Sing#, a modification on the C# programming language, which is compiled to Common Intermediate Language (CIL). The system incorporates a JIT-compiler, which compiles the CIL programs into programs in native code for the processor it is running on.

Instead of executing the operating system inside the interpreter as virtual machine, Singularity focuses on static checking and verification methods to ensure security. For example Singularity uses software isolation for processes, which works not by using the MMU or any other hardware features for isolation processes from each other. But it ensures at the JIT-compilation stage, that the processes are isolated properly. A so called Software Isolated Process (SIP) is checked against its manifest, and is restricted from loading any code inside itself or modifying its code. Communication between SIPs is done via an exchange heap, which allows for garbage collection of all SIPs isolated from each other.

Although Singularity is written in an interpreted language, it is actually executed as native code. One could argue, that it does not suffice the first level interpreted operating system definition, set by Tröger et al [23]. There are however some interesting approaches in the Singularity project. It dropped the performance and backwards compatibility priorities of conventional operating systems. Singularity focuses on a clear design over high performance. Also it did not include any compatibility to existing applications or drivers, which allowed for rethinking of these architectures, but had the consequence, that drivers and applications had to either be written from scratch or ported to Sing# line by line. Additionally Singularity implemented five garbage collection algorithms, which could be used as inspiration for garbage collection in PylotOS . Although being an interesting system, Singularity will not be outlined closer, as similar approaches of safety by bytecode execution are already chosen.

JX [8] is a research operating system written mostly in Java. It incorporates a small set of functionality, which is implemented as a microkernel in C as well as its virtual machine to run the Java processes. JX could be considered somewhat unique, as it has some OS-functionality in its C code, but for example implements its drivers in Java. JX also opts for a JIT-compilation process for performance reasons and compiles its processes before running any of them. JX is detailed in Section 3.6.

A very interesting similar approach is JavaOS, an operating system mostly written in Java [20]. The main principles for JavaOS development was, to implement only the main functionality required by the Java Development Kit (JDK) and to implement as much functionality as possible in Java. JavaOS uses device drivers written in Java, as well as networking, file system and the window system. Ritchie [20, pg. 30] found the performance to be acceptable and the accelerated development time to be a real eye-opener. A more detailed look into JavaOS can be found in Section 3.7.

Fuchsia is a relatively new operating system from Google. Announced in 2016 a prototype version has been released and documented [9]. Fuchsia and its kernel Zircon don't adhere to the standard POSIX-way and looking into it is therefore seen as beneficial. A very interesting part of the system is, that a set of languages is to be supported natively, like Dart, Swing and C/C++-applications. Interoperability of programs written in these different languages is done by using communication over so called channels. Section 3.8 describes the main concepts of the Fuchsia operating system.

3.2. The C subroutine library

This section will look into the C library, the newlib C subroutine library to be exact. This library can be used for bare metal development, i.e. writing software which runs on hardware without an operating system. A look inside the list of what functions or system calls are needed by newlib to 'work' yields a look inside a possible system design.

Processes are transparent for the C library in most parts, so the main focus in this section lies on the usability of the C-subroutine library calls to invoke device drivers and transfer of data between devices and the program running. Also interaction with other processes in the system and files is a part of the list of syscalls to be implemented by the OS developer.

The following system calls are needed by newlib for linking (from [19]). These are sorted by the potential use in an operating system.

- **File manipulation and drivers**
  - `close` close a file
  - `fstat` status of an open file
  - `link` establish a new name for an existing file
  - `lseek` set position in a file
  - `open` open a file
  - `read` read from a file
  - `stat` status of a file (by name)
  - `unlink` remove a files directory entry
  - `write` write to a file
- **Process Management**
  - `_exit` exit a program without cleaning up files
  - `execve` transfer control to a new process
  - `fork` create a new process
  - `getpid` get the process id
  - `kill` send a signal
  - `wait` wait for a child process
- **Misc routines**
  - `sbrk` increase program data space (`malloc` is based on this)
  - `isatty` query whether output stream is a terminal
  - `times` timing information for current process

3.2.1. Device drivers and files

The newlib library bases on the idea to use files as abstraction for devices, like most POSIXy operating systems do. For example when using stdout, the C-program actually writes the string to a file, the `stdout`-file. The kernel translates that to a command to the driver of the terminal or window-system.

However the kernel does not need any concept of files on a disk, i.e. it does not need any file system, but rather it needs to be able to work with integers representing a file handles. It also needs to be able to register a name with a functionality or file-handle.

The `open` call can easily be used to initialise the hardware or reset the hardware for a later writing or reading use. For character devices `read` and `write` can be used to read characters or bytes from and to write characters or bytes to the device. `close` could be used to deinitialise the hardware and to destroy the context, i.e. for a user program to signal, that it is finished working with that device for now.

Abstracting devices behind files is one possibility, which is aided by the C library for conventional POSIXy operating systems. However the C library does not contain functions for changing properties of the device, like the baudrate for a serial device or the bitrate for a video encoder. So programs needs additional System Calls (syscalls) for changing settings of the device.

In a matter of fact drivers for most of the operating systems have to fulfil an interface, which has entries like `write`, `open`, `close`, `read`. This makes it easier to write driver software, as the semantics of the calls are well defined within the scope of the C library, which the kernel developers can just adapt for their driver interface.

3.2.2. Process Management

Newlib also needs user provided functions to interact and manage processes. The `_exit` call has to care about cleaning up after a process is finished, i.e. either the program calls `exit` or the end of the `main`-routine is reached.

Also the POSIXy `fork` routine has to be implemented, to newlib is able to create new processes, `getpid` for reading the process it. Also signalling (see Section 3.3.2) is a part of the newlib model, as well as waiting for child-processes. It however makes no assumptions about the process model in the system.

3.3. 4.3BSD

This section handles 4.3BSD architecture, i.e. its driver architecture, start-up routines and process management, as well as interprocess communication facilities. 4.3BSD was released in 1986. It is therefore the oldest operating system examined in this thesis. It was released for the VAX machines by DEC. Although interesting, accounting and general architectural considerations are not outlined for spatial and relevancy concerns.

3.3.1. Process Management

A process is a program in execution [13]. A process in 4.3BSD operates either in kernel or user-mode, i.e. in a processor-mode, which allows executing privileged instructions or one which does not. A process can switch to kernel-mode by executing a system-call or syscall, mostly implemented by a trap into the kernel-code.

The resources of a process can be placed in two categories: the *user*-structure (u.) and the *proc*-structure, i.e. the user-mode resources and the kernel-mode resources. The user-mode resources are needed in memory only when the process is executing and may be swapped out of memory, when it

is not. The kernel-mode resources however have to stay in memory, as they are needed by scheduling or checking whether a process is awaiting an event, etc.

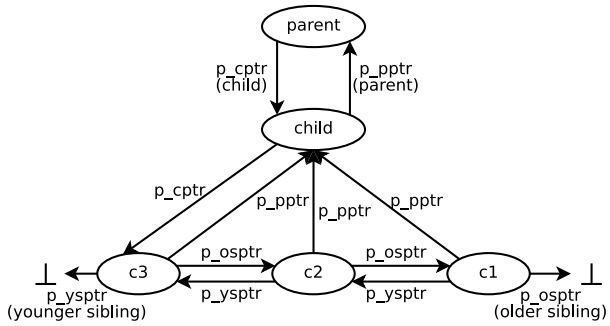

Figure 3.1.: Tree of processes in 4.3BSD [13]

The proc-structure of a process contains its kernel-mode resources and describes the process, its information for scheduling, memory management, event and signal handling, user- and group-identifiers and accounting. It includes priorities, nice and counters for CPU usage and sleeping time, the unique Process Identificator (PID), user-identifier for permission checking as well as pointers to the proc-structures of the parent-process, the siblings and children. The 'family'-relationship pointers form a tree, as depicted in Figure 3.1. The memory management facilities need information about the executable file (or a descriptor to it), a pointer to the page tables as well as its size, user area page tables and information where to find them, when swapped out.

Event processing and signals are a very important tool for the kernel to communicate information to processes. The proc-structure needs to have an entry for the event a process is waiting for, so the kernel does not have to swap in process information when an event occurs just to decide, whether a process state needs to be changed. Also information about the pending signals as well as the ignored and caught ones and its process group id need to be held. Accounting information, namely the `rusage` structure and quotas are needed also in the proc-structure. Real-time timers a process is waiting for to expire are also held in the proc-structure.

A process can be in one of five states, which influence whether a process can be scheduled to run or not. When a parent-to-be calls the `fork()` syscall to create a new process, the new process will be placed in the `SIDL` state, which is an intermediate state, showing that the process is not yet created completely. When done, it will be placed in `SRUN`, meaning it can be scheduled and can run. From there it can toggle between `SRUN`, `SSLEEP`, i.e. wait for an event, and `SSTOP`, so it's stopped by a signal or the parent, as often as it likes. From these three processes it can also be killed, landing in `SZOMB`, leaving it as a deceased zombie-process, which is waiting for the parent to be notified. When the parent is notified, the process is terminated.

The system also maintains several lists with the help of the proc-structure. There is the zombproc-list, which lists all zombie processes, allproc-list, which lists every process, which is not in zombie-state and the freeproc-list, listing all proc-structures, which are not in use. For scheduling there are two additional lists: the run-queue, which is used by the scheduling algorithm to create a schedule, and the table of sleep-queues (the event the processes are waiting for are hashed) to find the processes waiting for a given event.

The user-structure (u.) is allocated dynamically, may be swapped, when the process is not running. It is mapped to a specific address, when the associated process runs. It contains:

- user and kernel-mode execution states
- state related to system calls
- descriptor tables
- accounting information
- resource control information
- per process kernel stack

The execution state is defined by the hardware architecture, on VAX-machines it includes the general purpose registers, the stack pointer registers, program counter, processor state registers, segment base and length. There is also a software extension to what VAX calls execution-state, which is used by the kernel to store additional information, like the size of the page table, pointers used in context switching and segment copying as well as code which is used when delivering signals. The kernel stack is used for running kernel code, so when switching into the kernel-mode the stack is also switched to the kernel-stack.

4.3BSD allows sharing non-modifiable code between processes. Each segment of shared code is described in a text-structure. On an `execve` syscall, if the text-section is marked as pure (non-modifiable), it is loaded separately from the rest of the executable file (i.e. data and bss-sections) and a text-structure is created. Future loads of the same executable file may reference the same code.

Context switching 4.3BSD knows two types of context switching: voluntary context switching, i.e. the running process relinquishes from the CPU because it is blocked waiting for an event or releases the CPU itself (e.g. with the sleep()-system call). Involuntary context switches happen at the end of a time-slice, so the running process is switched to another process on the occurrence of a timer interrupt. While the first is synchronous to the currently running process, the latter happens asynchronously. This asynchronism is handled by hardware and transparent to the system.

Upon entering the kernel, the hardware execution state is stored on the kernel stack of the process. VAX machines have instructions for storing and restoring structures called PCB, which defines the context of a process on VAX, so switching the PCB also switches running processes. `swtch()` selects the process with the highest priority and calls `resume` with the correct parameters to continue executing that process (so actually doing the switching).

A voluntary context switch is done via a `sleep()` call with a wait-channel given[1]. The wait-channel identifies the event for which to wait. The system keeps so called sleep-queues of processes as a table of queues, the table indices are hashed wait-channels to find the waiting process(es) easily, when an event occurs.

`wakeup()` will awake all processes waiting on a wait-channel, i.e. will remove all waiting processes from the sleep queue waiting on that wait-channel. If a process is swapped out of memory, it needs to be swapped back into memory. If a process is in the SSTOP-state, it is kept in that state, until it is continued by either a `ptrace()`-call from a debugging parent, or by a CONTINUE-signal.

[1] Note: the `sleep`-syscall has changed over the years and blocks the calling process in modern Linux-systems for a given amount of time

Synchronisation When processes are sharing resources, which may only be accessed by a single process at a time, synchronisation is needed. Normally flags are used to synchronise processes. Synchronisation between normal processes is done by checking a flag, if it is set, it is marked as wanted and a sleep on the address of the shared resource is set in place. If the flag is cleared, it needs to be locked. Checking and setting the flag (locking), needs to be an atomic operation. After the process is done with the processing, which may not be interrupted by the other process, it clears the lock and checks the wanted-bit. If it is set, `wakeup()` is called to awake waiting processes.

A special case of synchronisation is needed when synchronising bottom and top halves of the system. The bottom half, i.e. interrupt service routines, may never wait for any event, as they cannot be interrupted nor pre-empted by a process. So flags cannot be used in this scenario, but the top half might have to prevent the bottom half from running in certain cases. It can raise the priority level of the processor or mask certain interrupt service routines, while accessing shared data structures or resources, and lower it afterwards.

The problem of deadlocks may arise when locking and waiting on flags, which is out of scope for this thesis.

Process Creation and Termination Creating a new process is done in 4.3BSD exclusively by a parent process, which uses the system call `fork()`, which will return once in the parent with the PID of the child for further referencing, and with the return value of 0 in the child's context. The child is an exact duplicate of the parent, except its PID.

The system allocates a new proc-structure for the child, duplicates the context of the parent for the child and schedules the child to run. The proc structure is then filled with the values of the parent (except the PID). The child therefore inherits the parent's privileges and limitations, opened files and other u.-area info like the User Identificator (UID), signal actions or its nice value.

Termination of a process can be done voluntarily (by calling the `exit()` system call) or involuntarily by receiving a signal, which terminates the process. The kernel will see both cases as a call to the internal `exit`, which will cancel pending timers, release virtual memory, close the process' descriptors and handle stopped or traced processes. Also the kernel will:

- remove the process from the *allproc* list and put it into *zombproc*
- change the process state to `SZOMB`
- set a flag, indicating no process is running at the moment
- set the termination status and accounting info in the proc structure
- notify the parent

The CPU is then rescheduled. Once the parent calls `wait()` or `wait3()` to search for deceased processes in the descendants, it will receive the child's exit status. The deceased child is then properly freed by putting the proc-structure back into the *freeproc*-list for reusing it for another process.

3.3.2. Interprocess Communication

Signalling

Signals only barely qualify as interprocess communication. They do not allow sending data, only signal certain events have occurred and allow the process to react to certain conditions. Also signals

enable the process debugging facilities of 4.3BSD to work properly. UNIX defines a set of signals for certain software and hardware conditions, which may arise during normal execution of software. In 4.3BSD signals may be seen as equivalent to hardware interrupts on a higher level, as they interrupt the normal process execution to indicate an exceptional state.

A process may send a signal to a process (with `kill()`) or to a process-group (`killpg()`, which sends the same signal to every process in a group[2]). Also the system may send signals to processes on certain hardware conditions, like an illegal instruction in the code of a process. Upon the delivery of a signal to a process, an action is invoked, either the user set a signal handler for the specific signal, or the default action is invoked. The default action may be the termination of the process, the termination with creating a core-file containing the complete state of the process, stopping the process, or the default action may be to ignore the signal.

The *SIGSTOP* and *SIGKILL*-signals cannot be ignored or caught by a user handler, this ensures that the system can always kill and stop runaway processes. A signal handler is a user-mode procedure to be called, when (a) specific signal(s) occur. Signals may be masked, so they will not be delivered with `sigblock()` (which adds a mask) or `sigsetmask()` (which sets a mask, replacing the old one). *SIGKILL*, *SIGSTOP* and *SIGCONT* (continue) may not be masked. When executing a signal handler, the currently handled signal is masked from being delivered until its first appearance is properly handled. There are more system calls in 4.3BSD which are important to handling signals:

- `sigpause()` - relinquish the CPU until a certain signal is received
- `sigreturn()` - return from handling a signal to normal operation.

In the UNIX virtual-machine model signals can be seen as equivalent to interrupts. A syscall is equivalent to hardware instructions, signal handling to ISRs or trap handlers, masking of signals to masking of traps or interrupt requests and the signalling stack to the separate interrupt handling stack on VAX machines.

Posting a signal occurs in the context of the receiving process, except when the process shall be stopped. There are two fields in the proc-structure of importance for signalling: `p_cursig` and `p_sig`. Posting a signal to the pending signals is a patchwork of special cases in 4.3BSD. Some of these are the following:

1. if the signal is ignored, nothing has to be done, return
2. add the signal to `p_sig` and do the implicit work associated with the signal (e.g. *SIGCONT* will remove all prior signals, which would stop the process)
3. if the signal is masked, nothing more has to be done, return
4. do the associated action immediately or arrange that the process does the action
5. according to the process state:
    - `SSLEEP`: the process awaits an event
        - if the action is to stop the process: set the process state to `SSTOP` and notify the parent with *SIGCHLD*
        - if ignored: remove the signal from `p_sig`
        - otherwise: place the process in the run queue

[2] Process groups are not relevant to the closer examination of 4.3BSD, so not discussed here in detail

- SSTOP: the process is being stopped (signal would stop again: nothing to do) or debugged (nothing to do until the controlling parent allows running the process)
  - if *SIGCONT*: put to SRUN or SSLEEP if the process was blocked before stopping
  - if *SIGKILL*: will always terminate the process next time it is scheduled
  - otherwise: the signal will need catching, but the process needs a signal for continuing (SSTOP → SRUN)
- SRUN, SIDL, SZOMB
  - if the process is not currently running: place an asynchronous system trap so the action is executed immediately when scheduled

Whenever a process returns from a syscall or from sleeping the signal flags are checked for new signals. If one is found pending, it is moved from the pending set (p_sig) to the currently delivered one (p_cursig). Then the psig()-routine is called to carry out the appropriate action if one is to be done. psig() handles two cases: produce a core dump (core(), exit()), or invoking a signal handler.

A signal handler is executed by first masking the delivered signal, then calling sendsignal(), which arranges for the signal handler to be called as soon as the process returns to user-mode. This action is machine dependent; the signal context is placed on the stack used for signal handling (maybe a different one from the normal run-time stack of the process). Then code inside the u.-area is called (so called sigtramp(), which calls the signal handler and sigreturn() to reset the process state and return from signal handling. Afterwards p_cursig is cleared.

Sockets and Pipes

4.3BSD features pipes as well as sockets. The pipe-mechanism is based on the socket-stack. Sockets are implemented via new system calls, besides the known file-specific system calls. However pipes as well as sockets are represented as file descriptors in 4.3BSD. The old system calls could have been reused, but they would have broken the datagram and packet-semantics of some sockets.

While pipes only transmit streams of bytes, a socket may transmit datagrams or packets of data, with or without a connection. There are other properties for communication listed in [13]. The main goal for the socket-interface was, that it is transparent for the user which protocol was used and whether or not the other endpoint was on the same machine.

Assuming, the reader knows how to use both pipes and sockets, a description of the usage is skipped in this thesis. Also memory management for sockets and pipes is not of interest in the PylotOS project, as memory management is done by the interpreter and the virtual Python machine only uses an object space.

3.3.3. Driver Architecture

4.3BSD differentiates between three types of devices:

1. block oriented devices
2. message oriented devices
3. unstructured / character devices

The block oriented device drivers are represented as a struct `bdevsw`, character / unstructured devices as a struct `cdevsw`, which are kept inside a table per type. Accessing a device is done with a device number, which consists of a major and a minor number. The major number is used as index for a driver in the driver-table and the minor number is interpreted by the driver and can be used to identify a specific device.

A driver on 4.3BSD contains routines for servicing I/O-requests (top half), interrupt service routines (bottom half) and routines for autoconfig and initialisation. Also there can be an optional dump-function for dumping the memory contents on disc when the system crashes for post-mortem analysis. The top half is called through syscalls from user applications and may block using the `sleep`-function. The bottom half cannot block and cannot depend on a specific state of the process stack.

I/O-Requests serve as buffers between the top half and the device itself. The ISR will get the next I/O Request of a queue, transmits it to the hardware and waits for an interrupt of the device indicating that the transfer is finished. The I/O Request queue is the only mean to communicate between the top and bottom halves.

The queues are shared between several asynchronous calls of the driver routines, i.e. accessing the I/O request queues needs to be synchronised in the driver. So the driver has to raise the processors priority level (disable interrupts), to prevent the processor from running the bottom half, when the top half accesses the I/O request queue.

Block devices do I/O operations from and to system buffers, which involves memory to memory copy operations, whereas raw devices use user buffers, which bypass the system cache mechanisms. There could be devices with both a block and a character access. User programs need to ensure consistency between the system cache data and the directly accessed data.

Block devices are represented in a structure `bdevsw`. It contains function pointers for the kernel code to interact with the driver. The struct `bdevsw` holds among others the following function pointers:

- `open` the user program calls `open` on a special device file. This in turn will call the `open`-function of a driver. The driver will prepare the device for I/O-operations, will check that the device was found in autoconfig state and check its integrity.
- `strategy` read/write operations. The system translates an I/O request to calls to `bread` or `bwrite`, which call `strategy` to load or write data. Takes a buffer as parameter.
- `close` called after the last client terminates. The semantics are defined by the device.
- `dump` called on a crash to write the values of the physical memory to disc. The priority level of the CPU will be set to the highest possible level, so the driver needs to poll for the state of the device, as all IRQs are disabled.

Character devices / unstructured devices Terminal printers and terminal drivers are the go-to examples of character devices. Drivers for character devices need to fulfil the following entries among others in its structure `cdevsw`:

- `open`, `close` - for initialisation and deinitialisation of the device (driver)
- `read` - read data from the device

- `write` - write data to the device
- `ioctl` - get and set parameters for devices, or perform operations other than read and write. The implementation differs widely from the device to device.
- `select` - check the device - are data ready for reading, is space left for writing. This entry point is meaningless for raw devices, because there are no buffers. This function will be called by the `select`-syscall.
- `stop` - stop the output on the device (useful for terminals only)
- `reset` - reset the device state after a bus reset. This entry point is used by the bus-adaptor. The driver resets and re-initialises the device and its data structures associated with the bus, for example it will reallocate the buffers and restart (maybe already begun) DMA transfers.

Autoconfig and probing Some devices may be present or not in different numbers on different memory addresses depending on the configuration of the system. The operating system needs to configure them as present or fail gracefully if they are not. In 4.3BSD there is a static configuration procedure, which is executed at compile-time, and a dynamic autoconfig phase on system boot-up (run-time).

For the static configuration files are used, which identify what devices may be present on the target system and also where in the memory their registers will be visible. The program `/etc/config` is used for parsing these configuration files and including the correct drivers in the bootable image.

The autoconfig phase in the system's boot-up consists of the drivers probing for devices on where they believe it could be placed. For every device which may be configured into the system, a probing-routine of the driver is called. The driver tests if the device is present and provokes an interrupt from the device. The system traps into the IRQ use the vector to associate the ISR of the driver to that vector. If provoking an interrupt is not possible, the system will assume, that the IRQ-vector, the driver returns will be valid and configure its ISRs accordingly. If the system does not receive an interrupt and the driver does not return an IRQ-vector, the device is assumed not to be present.

A driver may also need to probe for devices attached to a controller, so called slave devices. The `slave()` routine is called to indicate whether a slave device is present.

If a device is found, the `attach()` routine is called, so the device and the software state will be initialised. On discs this normally means, that the partition sizes and the partition table is read.

Device naming User programs use virtual files in the file system to access devices. The inode for these special files contains the major and minor number for the driver and device. The minor number for the device may change on reboot. Device files however point to the same device no matter where in the system it is placed.

3.3.4. System Start-up

Bootstrapping and system start-up of 4.3BSD is divided into four stages. First the system is bootstrapped by the *boot* program. It will load the system's kernel and start the main processor. Then the kernel takes over and goes through three stages of start-up, the first one written in assembly, the second one handles machine-dependent initialisations and the last one handles machine-independent actions. Figure 3.2 depicts these four stages.

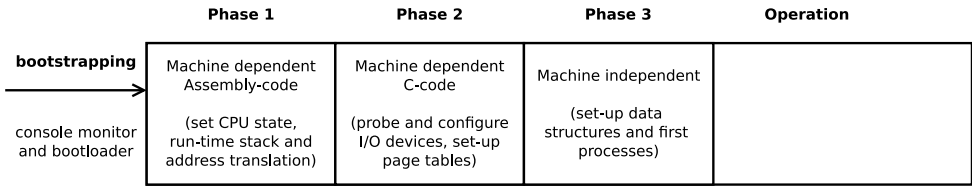

Figure 3.2.: Boot-up procedure of 4.3BSD

Bootstrapping

The bootstrapping-process is highly dependent on the machine. Most VAX-machines have a co-processor, so called console monitor or a console processor, which is mostly concerned with bootstrapping the system, i.e. loading the operating system kernel into memory and setting up the execution environment. Most console monitors however cannot use 4.3BSDs file systems, but have their own proprietary file system formats and secondary storage.

The console monitor loads the bootloader *boot*, which is able to use 4.3BSDs file systems. *boot* is generally used to load stand-alone programs, i.e. programs, which can run without the presence of an operating system. The *boot*-program can use parameters, which are placed inside of special registers. The first register contains flags on how to boot (register *boothowto*), and the seconds one contains the boot-device (*bootdevice*). Other architectures might place these information onto the run-time stack. Depending on the value of *boothowto* the bootloader will either load the file *vmunix* of the *bootdevice* or it prompts for the filename and the boot device and partition. The values in these registers can be modified with switches on the front of the machines.

boot loads programs to the address 0x00000000, so it has to relocate itself, before loading any new code in. Also the Central Processing Unit (CPU) is reset as well as its I/O subsystems. The interrupt level of the CPU is set to the highest possible value, so no interrupt can pre-empt the start-up process and virtual address-translation is disabled. Then the CPU is instructed to run the loaded code.

Step 1: Machine dependent low level Initialisation

The first stage is comprised of assembly code, which is necessary for running C-code in the two later stages of start-up. There are several actions to take care of:

- set-up of the run-time stack
- identifying the type of CPU
- calculating the amount of physical memory in the machine
- enabling virtual address translation
- initialising the MMU
- crafting the hardware context for process 0
- invoking the initial C-based entry point of the system (*main()*)

Identifying the type of CPU is important as not every processor will support every instruction of the VAX architecture, so some might need to be emulated by the kernel. Also I/O architectures might differ from machine to machine, as VAX only defines the processor, but not the I/O subsystems. For

identifying the system, the *system identificator register (SID)* is used, which is pre-loaded with well-known values, which identify the system.

The system identificator is also used to determine the console monitor and therefore the interface to its functions, decide on the devices that might be configures, handle exceptional machine conditions, calculate the real-time delay implemented with spin loops and decide on how to set-up the interrupt handlers, so a single 4.3BSD image can support several VAX machines.

Step 2: Machine dependent initialisation

The *main()* routine takes one parameter: the number of the first available page. This value is calculated from the size of the kernel, pages with run-time stack and data structures. *main* calls *startup* with the same parameter for the machine dependent initialisation. It will:

- initialise the error message buffer
- allocate memory for system data structures
- initialise the kernel memory allocator
- autoconfigure and initialise the I/O devices

The message buffer is a 4 KB circular message buffer at the top of the physical memory. Diagnostic output is stored inside the message buffer.

Autoconfigure As shown previously in Section 3.3.3 the I/O devices are probed and detected by autoconfiguration. The system knows beforehand which devices might be present, and therefore has two tables of devices, one for the MASSBUS and one for the UNIBUS.

In the autoconfigure-step, every possible nexus-location is probed, for an active connection to the system bus. If there is a device present the type of the device is interpreted and the device is initialised and configured. For adaptors the devices on the secondary buses need to be identified, for which the appropriate table is used. The driver (for UNIBUS devices) or the system (for MASSBUS devices) will probe for the device, initialise it and if this is successful, the device is then a so called configured device and can be used.

Another problem to take care is the registration of ISRs. The *probe* routine will instruct the device to generate an interrupt. The default ISR will save the vector and the priority level to variables, which can then be picked up by the system, when it returns, to set the correct interrupt service routine to the correct interrupt vector. Non-configured interrupt vectors will be left with the default handler, which, after the autoconfigure-step, will print a diagnostics message and ignore the IRQ. Drivers cannot be loaded at run-time by 4.3BSD.

Step 3: Machine independent initialisation

Step 3 sets up the first processes, namely process 0, 1 and 2. Process 0 is marked as running (its run-time memory where created in step 1). The last page of the run-time stack is set as read-only (so called red-zone), so it cannot expand over the memory limits without causing the system to trap. Its structural information like UID and Group Identificator (GID) will be inherited by any new process.

The paging system is initialised, then the disk quota system and then the real-time clock, i.e. a device, which interrupts the CPU in regular intervals. Then the network memory management system is, terminal I/O facilities, network interfaces and communication protocols, the process management data

structures, text tables, inode cache and inode tables, swap management and the file-system name cache are initialised.

Before starting additional processes, the root file-system is mounted. Then process 1 (executing */etc/init*) and process 2 (copies pages of memory to secondary storage) are started. The time of day is consistency checked with the last-modification date in the superblock of the file system. Recalibration of the time of day can be done in the user-level. After process 2 is created, *sched()* is called inside the *main()* main-function, which will never return from the context of process 0.

User level Initialisation File system consistency checks and repairing damages of crashes in done in user level, as the kernel only care for reading and writing files. If problems are detected, the system needs to reboot and repair its file system. Additional file systems are mounted, devices for swapping and paging enabled, and several background daemons are started, including cron, a process for system accounting and syslogd.

System Shutdown The console monitor is necessary for the ability to reboot. It can give *boot* necessary parameters, so it can decide what to do. 4.3BSD has a system call *reboot*, which takes one parameter. This parameter is a superset of flags, which also *boot* takes. These include flags to halt, reboot into single-user and multi-user mode, dumping a crash-dump before rebooting or disabling the data in the buffer cache to be written to disk (for example if it may be corrupted).

In case of a catastrophic failure, the *panic* routine will write a crash-dump and reboot the system. Upon reboot, the file systems are checked for inconsistencies and the system is rebooted once again to resume normal operation. If a crash-dump is found on reboot, it will be copied to a file in the file system alongside the vmunix-file for debugging purposes.

3.4. Linux 2.6

In this chapter the driver architecture of Linux 2.6 is discussed, first interrupt handling and then the inner workings of the driver interface and architecture is outlined. Process management of Linux is not elaborated.

3.4.1. Handling Interrupts and Exceptions

Handling Interrupts is essential for dealing with hardware, as many devices work interrupt driven or servicing them by polling their state is tedious or would have a great impact on the performance of the overall system. Bovet and Cesati [3] focus on Linux 2.6 on Intel x86 architecture, most concepts however can be found on ARM hardware as well, so the strategies presented in [3] could be reused for ARM.

An exception is bound to a process, meaning that the process running is the reason of the exception. The process may need some servicing, maybe needs to be aborted after a division by zero-exception or just wants the kernel to execute a command (syscall) by issuing a supervisor-call / software-interrupt / syscall-instruction. An IRQ on the other hand is not bound to a specific process, so an ISR could run in any process context. Linux interleaves the execution of interrupt service routines, so that an interrupt can occur and can be handled, while another interrupt service routine still runs (but finished acknowledging the interrupt).

Linux sets up the Interrupt Descriptor Table (IDT) in two rounds. First it replaces the values set by the Basic Input Output System (BIOS) with `ignore_int()` routines, to ignore the interrupt completely.

Then it starts to replace those with meaningful interrupt handlers.

All ISRs have to save the vector and the current values of the registers to the stack, acknowledge the interrupt to the PIC to allow further interrupts, execute a handler function associated with that vector, i.e. with all devices that may raise interrupts over that specific interrupt line and terminate by jumping into the function `ret_from_intr()`. An ISR may never block or execute a statement that could potentially block, as this would lead to a frozen system.

Bovet and Cesati [3] distinguish between three types of interrupts:

1. I/O Interrupt - a device needs attention, the ISR queries the device to determine the proper course of action
2. Timer Interrupts - a timer has gone off, either an APIC-timer or an external one
3. Interprocessor Interrupts - a CPU issues an interrupt to another one

The interrupt vector the PIC presents may not tell the complete story however, so the (first level) ISR needs to be flexible enough to handle several devices. There are two reasons for that:

- Interrupt line sharing - the line of the PIC is shared by many devices. The ISR needs to query all devices for whether they need service, if so, handle the interrupt per device.
- Interrupt line dynamic allocation - some devices do not allow sharing interrupt lines, but the kernel may still want to use one interrupt line for these devices. The vectors will be bound at the 'last possible time' to the device, e.g. when the user mounts a floppy volume.

The actions needed for the attention a device may need can by divided into three categories. Critical actions can be executed quickly and must be performed as soon as possible, so they are executed immediately in the interrupt handler with maskable interrupts disabled. These are for example acknowledging the interrupt to the PIC, reprogramming the PIC or the device controller or updating data structures that are shared by the device and CPU.

Non-critical actions include updating data, which are only accessed by the CPU, can be done quickly and will also be executed immediately inside the ISR, but with maskable interrupts enabled. Non-critical deferrable actions can wait for a long time without affecting the kernels operation. The process interested in the data needs to wait until the deferred action is executed. These could include copying buffers into a process' address space.

An ISR should defer as much work as possible to allow that other ISRs can be nested, i.e. the deferred work can be interrupted and other interrupt handlers can be executed, 'while' the pre-empted ones executed non-critical code.

The kernel maintains an array of `irq_desc_t` structs, as shown in Figure 3.3, for finding the appropriate action, when an interrupt is issued. This array stores a list of `irqaction`s pointing to a handler function for that specific vector per interrupt line. Also a PIC-object (struct `hw_irq_controller`) is stored in the struct `irq_desc_t` because it is possible for a different system to have a different interrupt controller or a system may have several interrupt controllers for several line-ranges. So the PIC-object serves as abstraction from what type of interrupt controller is used.

The PIC-object has the following methods:

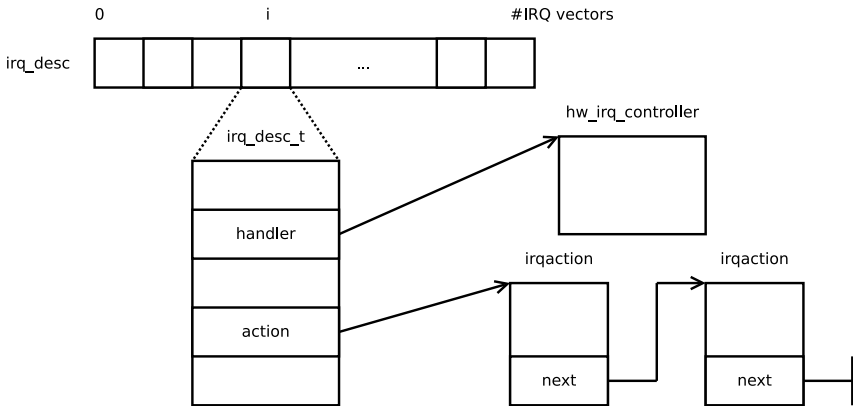

Figure 3.3.: Data structures used by the generic ISR [3]

- `startup`
- `shutdown`
- `enable`
- `disable`
- `ack`
- `end`
- `set_affinity` (on multiprocessor systems)

These allow to start up and shut down lines on the PIC, as well as enable or disable them (same operations on a standard PIC, but different on APIC. `ack` acknowledges an interrupt by sending the appropriate data to the interrupt controller, and `end` is called when an ISR terminates. `set_affinity` allows setting the 'affinity' of an IRQ, i.e. what CPUs may respond to a specific request.

`irqaction`s save a handler function to be called when a specific interrupt vector is raised. With the list of `irqaction`s per `irq_desc`-field it is possible to share interrupt lines between devices as the handler functions for every associated device is called. That means, that the kernel will execute every `irqaction`, which is associated with the interrupt vector. With the handler functions the `irqaction` stores a flag, which may specify the following:

- `SA_INTERRUPT` - the handler runs with interrupts disabled, the kernel will disable maskable interrupts, before the function is called.
- `SA_SHIRQ` - the device allows line sharing
- `SA_SAMPLE_RANDOM` - the interrupt can be used to source random events for the random numbers generator

At the start of the system all processors will have an equal number of interrupts lines to take care of. Later on, the kernel thread `kirqd` re-balances the interrupt load by using interrupt counters of the processors and also corrects wrong associations.
A generic system handler function saves the process context to the stack when an interrupt occurs, i.e. the register contents. The interrupt vector is determined and saved.

The system may switch to a kernel stack for ISR-execution. System-code will acquire the spinlock to prevent other processors from handling the same interrupt concurrently. Then the interrupt is acknowledged and silenced at the PIC. It may be raised again when re-enabled later on. Then checks are executed, whether the same interrupt is handled by another CPU, if so, work is deferred to that CPU.

The handler is executed in a loop, so when the same interrupt occurs again while the interrupt handler runs, it can run again to also handle the second one. When the loop is finished, the `end`-method

is called of the PIC-object, to re-enable the IRQ line or acknowledge the interrupt on an APIC if not done already.

The actions will be given the interrupt vector, the device identifier and a pointer to the structure containing the original register values pushed to exception stack on ISR entering. The first two allow an action to potentially serve several interrupt vectors or devices, but they are mostly unused in practise.

Each of the actions returns 1 if the interrupt was recognised, i.e. was issued by a device it controls. This allows the kernel to update its data structures of unexpected interrupts and disable the line if too many unexpected ones are issued.

When the action was executed, the stacks are switch backed, and the call `ret_from_int` is used to restore the process state and return form the ISR.

Line sharing Some devices may not be able to share an interrupt line, but the kernel may still want to use them with a shared line. The system will then activate the interrupt at the last possible time. For a floppy disk volume, the according interrupt action will be set, when the volume is mounted.

On a mount of the volume, i.e. the first access to the device file, the driver may request the interrupt line by calling `request_irq`, which returns an error number if the line is already in use by another device. On unmount (or when the device file is closed) the driver requests the IRQ line to be freed with the function `free_irq`.

Softirqs and Tasklets are work packages, which are deferred to execute at a later time. They can be used for functions, which are important, but can wait. This ensures, that the ISRs can be very short and the real work can be done at a later time, when interrupts are enabled, so short IRQ acknowledgement times are feasible.

A softirq has to be executed on the CPU it was activated on. They can run concurrently on several CPUs, so they need to use a spinlock if necessary. Tasklets are implemented on top of softirqs and are executed in serial by the kernel, so they don't have to be re-entrant.

There are these four kinds of actions on these deferrable functions:

1. Initialisation — define a new deferrable function (when the kernel initialises itself or on module load)
2. Activation — marks a deferrable function as pending, it is executed the next time the kernel schedules a round of deferrable functions
3. Masking — disables a deferrable function, so not executed even if they are activated
4. Execution — execute pending deferrable functions (at well-defined times)

Linux uses a vector of 32 `softirq_action`-structs, of which only the first 6 are effectively used. It contains an action pointer, which points to the softirq function to be executed, and a generic data-field, which may be used by the function. A bitmask of flags, which indicate, whether softirqs are ready to be executed is stored per processor. Tasklets are softirqs with the lowest possible priority.

For initialisation of a new softirq the function `open_softirq` is used, which takes the index of the type, a function pointer and a pointer to the block of generic data. `raise_softirq`, given the id,

activates the softirq. The kernel checks regularly if softirqs are there to be executed.

The handling of softirqs does not differ much from handling of interrupts in interrupt context. At the end of the generic system interrupt handler a check is performed whether softirqs are ready to be executed. If so, it executes a set amount of softirqs.

3.4.2. Linux I/O Architecture and device drivers

Linux' I/O Architecture and its virtual file system are very much entwined. This thesis will not however detail the inner workings of the virtual file system, but work with it as a black box. The virtual file system depends largely on the device driver architecture and device drivers to carry out the functions called by user applications like `write` to a file.

The kernel uses the virtual file system to allow user applications to access kernel data structure. The devices and drivers are provided in the `/sys`-folder by sysfs.

Data Structures of the Device Driver Model

The device driver model of Linux consists of several garbage collected data structures.

The `device` object represents devices and contains (among others) the following attributes:

- `node` list of sibling devices
- `bus_list` list of devices on the same bus type
- `driver_list` pointer to the next device on the drivers list
- `children` list of children devices
- `parent` parent device
- `bus_id[]` position of the device on the bus
- `bus` the hosting bus
- `driver` pointer to the driver object of the device
- `power` power management information
- `detach_state` power state to be entered, when the driver is unloaded
- `release` freeing the device descriptor

Devices are globally stored in the `devices_subsys` list. A device is a child of another device if it cannot function properly without the parent device, e.g. on a PCI-based system a PCI-to-USB-bridge is parent to all USB-devices.

In fact the device driver model is based upon several lists. The `device`-objects are connected in a list of siblings and one of children. They are organized in lists by bus-type and driver. They also have a pointer to the parent element. A tree of children-sibling-relationship is formed with these pointers and lists.

The `device_driver` object represents a device driver. It contains the following fields (not a complete list):

- `bus` pointer to the bus descriptor hosting the supported devices
- `devices` pointer to the first element of the drivers device-list
- `owner` identifies the module, which implements the driver
- `probe`, callback method for probing a device
- `remove` `shutdown` callback functions, called when the device is removed or shutdown
- `suspend`, `resume` callback functions, called when the device is put to lower power mode / put back to normal power mode

The `probe`-function is called, when a bus driver detects a device, which could possibly be managed by a driver. It does further checks to determine if the device is in fact supported.

Bus objects are represented by the `bus_type`-structure. It contains:

- `name` name of the bus
- `drivers`, `devices` list of drivers and devices
- `bus_attr`, `dev_attr`, `drv_attr` pointer to objects containing attributes and methods to export to the sysfs for the bus, devices and drivers
- `match` callback function. Checks whether a driver supports a given device
- `hotplug` callback function. Invoked when a device is being registered
- `suspend`, `resume` callback function for changing the power state and saving device context

Bus objects are stored globally in the `bus_subsys` list. `match` is called, when the kernel wants to know if a detected device is supported by a driver. This function is usually very simple and checks the ID of the device with the supported devices list of the driver. `hotplug` is called when a new device is added to the driver model to set-up bus specific information for user level applications.

Device Files are the main mean to access devices from user level in Linux. Device Files are special files, which can be used like regular files with the same syscalls, and can be regarded as container for sequences of bytes. Linux distinguishes two type of devices, but does not give a precise classification [3]:

- character device cannot be addressed randomly (like a sound card) or the access times depend heavily on the location of the data on the device
- block device data on the device can be accessed randomly and the access times are small and roughly the same

The classification leaves out network interface cards, which do not fit in any of the two descriptions.

An inode contains all the information the file system needs to handle a file. Each file has its own inode, which the file system uses to identify the file. The inodes of device files do not include pointers to the contents of the file, but rather identifiers of the device, typically the type and a pair of numbers (major and minor number). The major number identifies the device type, the minor number identifies a single device in a set of devices, sharing the same major number. Character devices and block devices are enumerated independently.

In Linux 2.6 the major number is 12 bit, the minor 20 bit long. Both are contained within a 32 bit `dev_t`-type. They are also dynamically assigned. Each driver can request a range of device numbers at its registration phase (later in this section).

The user-mode application `udev` populates the `/dev` directory on start-up. It traverses the `/sys/class`-directory and looks for `dev`-files containing the major and minor number of a device. Then it create the corresponding device files in `/dev`. When a new hot pluggable device is detected the kernel will create a new process calling the `/sbin/hotplug`-skript.

File Objects are objects, which are created on calling `open` on a file. Reading regular files leads to blocks being read from disk, reading inode structures from the file system and locating the file, whereas device files could just be reading a temperature value of a sensor. The virtual file system needs to hide the differences from user land applications.

This thesis does not go into detail about file objects, but they have a table of operations possible on the file. Appendix A lists the file operation callbacks and their parameters.

File objects are created on `open` from inode information. The name is resolved, the inode object is created and the file operation table is filled with default functions for character or block devices respectively. Then the attached `open`-operation is called, which may modify the table of operations and set the operations to the ones specified by the driver. Syscalls from user applications are passed through to this table of operations.

Device Drivers

Device drivers are a set of kernel-mode functions to implement or respond to the programming interface defined by the virtual file system. Device drivers are supposed to control a device with these functions. The level of support a driver gives to the user application can differ dramatically. There are two stages of initialisation of a device driver.

Device Driver Registration is done in the first possible moment, so a user application can use the driver's functionality. A new `device_driver` descriptor is created and inserted into the device driver model as well as creating links to device files.

If the driver is built into the kernel, device driver registration is done in the kernel initialisation phase at boot-up, otherwise when the module is loaded. An advantage of building a module is, that the module can be unloaded and de-registered.

The kernel checks for supported devices of the driver on registration. It relies on the `match`-function of the `bus_type` and the `probe`-function of the driver.

Device Driver Initialisation is done is the last possible moment, as it allocates precious system resources, e.g. interrupt lines for devices, which do not support sharing.

There is a schema for obtaining resources, so driver don't allocate resources redundantly. There is a usage counter indicating the number of file objects referring to a device file, which is incremented in `open` and decremented in the `release`-method. The `open`-method checks the usage counter before incrementing. If it is 0, resources need to be allocated first. `release` checks it after decrementing. If it hit 0 deinitialisation can be done, i.e. resources can be freed, DMA channels and IRQ lines released.

Monitoring I/O operations

A driver needs to know whether an I/O operation is complete or whether it did fail. This is especially useful when it is done asynchronously.

Driving the device in polling mode [3], and repeatedly reading the status register(s) of the device is one way. So the driver will know, when a state change occurs, i.e. when the I/O operation had an effect. This technique is used for devices which cannot be interrupt driven. For longer operations the driver can yield the processor and hope to get scheduled later for re-checking.

The interrupt mode on the other hand is only possible if the device supports issuing interrupts. An example is given for reading data from a character device [3].

1. acquire the semaphore to ensure that no concurrent access is happening
2. clear a shared variable (with the ISR), which indicates whether an interrupt occurred
3. issue a read command to the device
4. wait until the interrupt-flag is set
   - when the interrupt occurs, the ISR will set the interrupt-flag and unblock the thread
5. copy the character to the output buffer
6. release the semaphore

3.4.3. Levels of kernel support for a device

The kernel can have one of these levels of support for a device:

1. No support - the user applications have to access the hardware themselves, by issuing `in` and `out` assembler commands themselves.
2. Minimal support - the kernel does not recognise the I/O device, but does recognise its I/O interface. User programs can treat the interface as a sequential device capable of reading and writing sequences of characters.
3. Extended support - the kernel recognises the I/O device and handles the I/O interface itself. There might not be a device file for a user app to work with.

In a 'modern' Linux there are many examples for the three levels. The X-server uses the no support level. It asks the kernel with the functions `iopl` and `iopen` to allow access to certain I/O ports. This

is efficient, but there is no interrupt support. In 'recent' Linux versions the kernel presents the framebuffer of a graphics accelerator as `/dev/fb` as abstraction [3].

Minimal support is granted to devices connected to a General Purpose Input Output (GPIO) type connector, like a serial connection or the parallel port. The kernel presents these interfaces as device files, but the application has to implement the protocol to manage the connected device itself by reading and writing from / to the device file.

The virtual file system does not give full access over a device. Special commands cannot be issued with `write` or `read`. Therefore the `ioctl` function was introduced. It receives the file object of the device file, a 32 bit parameter indicating the request type and an arbitrary number of parameters.

3.5. Windows 2000

This section describes the I/O system of Windows 2000 [21]. As with Linux, it will first describe the process of handling interrupts and then go into the details of the driver interface and procedures. This section focuses on the x86 architecture.

3.5.1. Interrupt Handling

Like Linux 2.6 Windows 2000 distinguishes between asynchronous interrupts (like hardware interrupts, timers) and synchronous exceptions (like syscalls or device by zero errors).

The kernel creates a trap frame on the current process' kernel-mode stack and store a subset of the complete process context in it, when an interrupt occurs. Upon leaving the ISR this context is restored, so the execution of the pre-empted process can continue. The kernel registers a generic ISR to the PIC. It stores and restores the process context and calls driver defined ISRs to handle the occurred interrupt. An unexpected trap leads to the system calling `KeBugCheckEx`, which halts the computer to avoid data corruption.

The interrupt controller uses the interrupt vector as index into the Interrupt Descriptor Table (IDT) to call a function. Windows 2000 fills this table on system boot with kernel functions to handle the interrupt or exception. Every processor has its own IDT, as it might be necessary to call a different function on the same interrupt. For example the clock interrupt will be necessary on all CPUs, but only one CPU would need to update kernel time information. The others might still need to reschedule or count thread quanti. The Hardware Abstraction Layer (HAL) selects the algorithm to balance interrupts between the processors. It normally selects the one, which handled the last IRQ of the same type.

Interrupt Request Level (IRQL) are a kind of priority for different interrupt sources. The kernel defines these levels in software and the Hardware Abstraction Layer (HAL) maps IRQ vectors to Interrupt Request Levels (IRQLs). An ISR with a higher IRQL can pre-empt a running ISR with a lower level. The kernel raises the IRQL upon entering an ISR to its level. Every CPU keeps its own IRQL, so a CPU might allow certain interrupts, whereas they are masked off by the IRQL mechanism on another one. The user-mode always runs in the lowest level. Only kernel code can raise or lower the IRQL.

As writes to the IDT might be costly, it is not modified upon raising the level. When a lower interrupt occurs, the kernel code will check the IRQLs. Then it will mask off the interrupt line, therefore disabling it until the IRQL is lowered below the level of the interrupting line, re-enabling it in the process.

A bus driver detects all devices connected to its bus and reports them as well as their resource (and IRQ line associations) requirements to the Plug and Play (PnP) Manager. The PnP Manager decides which device will get which interrupt line and which IRQL. On a uniprocessor system the association is done by 1:1 mapping, i.e. higher interrupt line will result in higher IRQL, on a multiprocessor system a round-robin association is used. Windows 2000 has no facilities to decide what device or interrupt gets which IRQL.

The system defines some IRQLs, in descending order:

- high - used when the kernel is halting the system and masking out all interrupts
- inter-processor - used when requesting another CPU to do some action like TLB updating
- clock - used for system clock interrupt. Track time of the day and do performance measurements
- device<n> - several IRQLs for prioritising device interrupts
- Deferred Procedure Call (DPC)/dispatch and Asynchronous Procedure Call (APC) - used for software interrupts, which drivers and the kernel can generate. In case of APCs, the user-mode might have created that interrupt. APC-level is lower.
- passive - normal thread execution

All levels above the DPC/dispatch-level face some restrictions. They cannot wait on objects, as that would need scheduling for transferring control to another thread. Also only non-paged memory can be used, as using paged memory, which is swapped away would lead to a page fault and in turn scheduling for context changes e.g. to the idle thread, while the page is swapped back in memory. If either of these restrictions is violated the system will crash, which is a common issue in Windows 2000 device drivers [21].

Interrupt objects represent ISRs to drivers. They allow drivers to register and identify ISRs. Interrupt objects contain the IRQL, the entry of the IDT, the address to the ISR and the IRQ line. The functions `IoConnectInterrupt` and `IoDisconnectInterrupt` allow for connecting (enabling) or disconnecting (disabling) an ISR with a particular IDT entry. These can be used by the driver while loading / unloading to set or clear their ISRs. These functions hide the details of the interrupt hardware and the structure of the IDT from device drivers.

The interrupt objects allow daisy-chaining ISRs together if they allow line sharing (the PnP Manager will make sure that device requirements in terms of line sharing are fulfilled). That means several ISRs can be executed one after the other to service their devices if their devices all issued interrupts to the same interrupt line.

If one of these ISRs reports back a status to the dispatcher indicating that the IRQ has been served, no other ISR will be executed. If any other device also issued an interrupt to that line, the processor will be interrupted once again.

Deferred Procedure Calls (DPCs) are system tasks, which are not as time critical as the currently executed task. Device drivers use DPCs to complete I/O requests. The ISR will do the bare minimum work to acknowledge the IRQ and defer most of the other work to a later point in time. DPCs run in DPC/dispatch IRQL, so other device IRQs may occur and may pre-empt the DPC-execution.

DPCs are represented by DPC objects, which are stored in a DPC queue for each processor. These objects are not visible to user level applications. They store most notably the address of the system function to call.

The system will normally put new DPC objects at the end of the queue and associate them with the processor which adds them. This behaviour can be modified in Windows 2000. A driver may choose to add a targeted DPC, so it will be added to the DPC-list of another CPU. It may also add a DPC with a higher priority.

When the system adds a DPC to the queue, it can have different behaviours. When the DPC is targeted at the current CPU and the priority is high or medium, a software interrupt is issued immediately (which is probably masked off by a higher IRQL at this moment), if the priority is low, a software interrupt is only issued if a certain threshold of elements in the list is passed. If the DPC is targeted at a different CPU and the priority is high the other CPU will be interrupt by the inter-processor interrupt mechanism to force it to execute its DPC queue. Otherwise a threshold needs to be passed.

When the system's IRQL drops from something above the DPC/dispatch level to a lower level the system will be interrupted by a waiting software interrupt for DPC-execution. That means the system will begin executing DPCs. Only when the queue is drained (empty) will the IRQL drop below DPC. Figure 3.4 depicts the procedure for executing DPCs in Windows 2000.

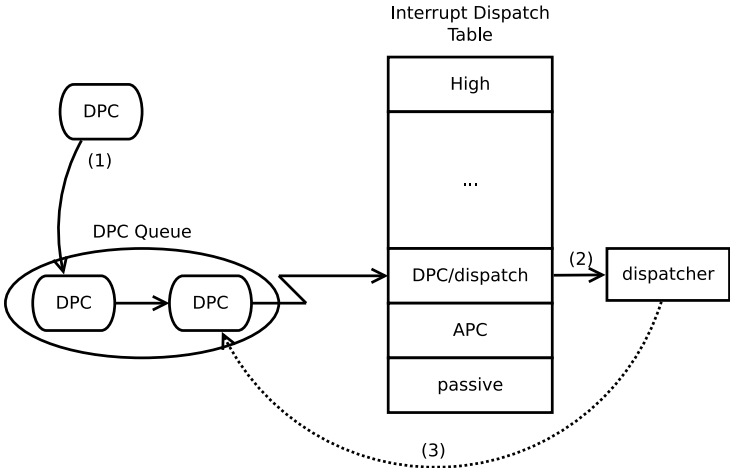

Figure 3.4.: Procedure for executing DPCs [21, pg. 109]
(1) A timer expires, the kernel queries a DPC to release every thread waiting for the timer. A software interrupt is issued (stored by processor until IRQL drops).
(2) IRQL drops, the software interrupt occurs. Control is transferred to the (thread) dispatcher
(3) The dispatcher executes each DPC routine until the DPC queue is empty. If required it also reschedules the CPU.

When a DPC-command is executing it cannot assume anything about what the process address space might look like. It must not page fault and cannot use system calls, but it can call kernel functions. It can also access non-paged system memory as that is always mapped.

3.5.2. I/O System

Windows 2000 manages drivers differently from Linux. Several components of Windows are involved with handling drivers and devices. The I/O Manager sends commands to device drivers and will be notified by the driver when a request is completed. Drivers may also use other drivers or devices for

their service, which they only can do through the I/O Manager. A driver provides an I/O interface for a particular type of device.

The PnP Manager works together with the I/O Manager and bus drivers to load device drivers for devices which are added at run-time. The Power Manager works together with the I/O Manager for guiding the system or individual devices into different power states. The registry database stores a description of basic devices, which are attached to the system, driver configurations and settings. The HAL insulates drivers from the specifics of the CPU or interrupt controller by providing Application Programming Interfaces (APIs) that hide differences between platforms. HAL is the bus driver for all devices on the motherboard that are not controlled by other drivers.

Figure 3.5 depicts the procedure of accessing a hardware device from a user application. A user level application uses a user-level API and traps into the operating system kernel. The kernel directs the request to the I/O system services, which in turn hands it over to the I/O Manager. It forms an I/O Request Package (IRP) to send to the appropriate driver(s). The driver may use support routines from the operating system kernel and the HAL to access the I/O ports and registers.

Appendix B shows the structure of an IRP and describes its fields. This structure definition is newer than Windows 2000, it may have changed since Windows 2000.

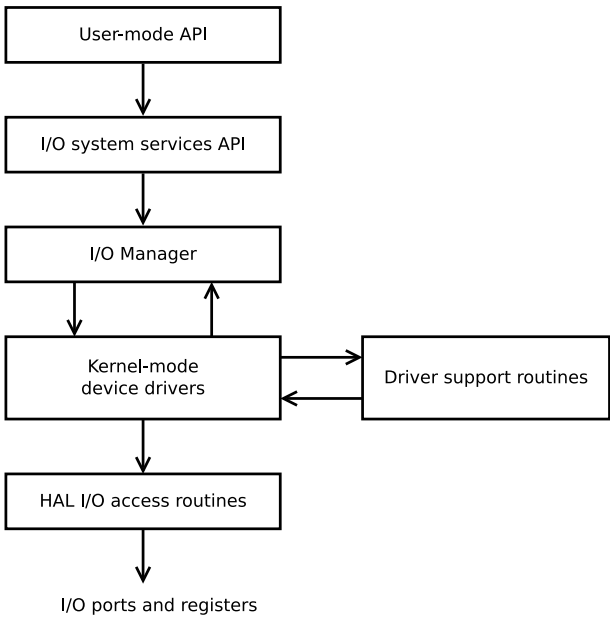

Figure 3.5.: High level view of the procedure to access hardware in Windows 2000 [21]

User applications use documented functions like `read` or `write` to access virtual files, which represent a device. Accesses to these device files are redirected by the I/O Manager to call the respective driver. The data within these virtual files is regarded as a simple stream of bytes.

3.5.3. I/O Manager

The I/O system is packet driven. The I/O Manager describes a model in which I/O requests are delivered to drivers in the form of IRPs. That allows an application thread to manage multiple I/O requests concurrently.

The I/O Manager creates an IRP, fills it with information, passes a pointer to it to the drivers and deletes the IRP upon completion. The driver fulfils the action specified in the IRP and hands it back to the I/O Manager for usage in other drivers or for deleting the packet. The I/O Manager also contains code which is shared between drivers, like calling the functionality of other drivers, IRP queue management and timeout support.

The drivers provide a uniform interface, so the I/O Manager can call any driver without knowledge of the structure of the driver or the functionality of the device.

Device drivers need to conform to a set of implementation guidelines specific to the device type and the role the driver plays in managing the device. Windows 2000 defines several different kernel mode drivers:

- File System drivers accept I/O requests to files and request I/O to mass storage and network devices
- Windows 2000 drivers integrate into the Windows 2000 Power and PnP Manager
- Legacy drivers were written for Windows NT but run unchanged in Windows 2000. They lack support for Power and PnP Manager and might therefore limit the system's PnP or power management capabilities.
- Win32 subsystem display drivers translate device independent graphics requests to device dependent ones. A display driver implements drawing operations by either writing into the framebuffer directly or by communication with the graphics accelerator.
- Windows Driver Model (WDM) drivers adhere to the WDM, which includes support for Windows 2000 PnP and Power Manager. These drivers are source-compatible between Windows 98, ME and 2000. It defines three types of drivers:
  - bus drivers manage physical or logical busses (like PCI or USB), can detect new hardware and inform the PnP Manager about the new device and allow setting power states for the bus
  - function drivers manage a particular device. The bus driver presents the device through the PnP Manager to the function driver. It exports the operational interface of the device to the OS and has the most knowledge about the operation of the device.
  - filter drivers are logically above or below the function driver, augmenting or changing the behaviour of a device or another driver

  The bus driver assists the PnP Manager in enumerating devices on the bus, detecting membership states, sometimes power management on the devices on the bus and accessing bus specific registers. Function drivers are the only drivers to access the device itself.

Windows 2000 also defines user-mode drivers (virtual device drivers and Win32 subsystem printer drivers), which are considered in this thesis.

Figure 3.6 shows a layered stack of drivers, in this case a file system driver and a disk driver. The file system driver receives an I/O request for a specific location (offset) in a file. It then computes a logical location on the disk and sends an I/O request to the disk driver via the I/O Manager. The disk driver calculates cylinder, track and sector to manipulate the file on the disk. It sends the request to the disk (maybe by using another driver).

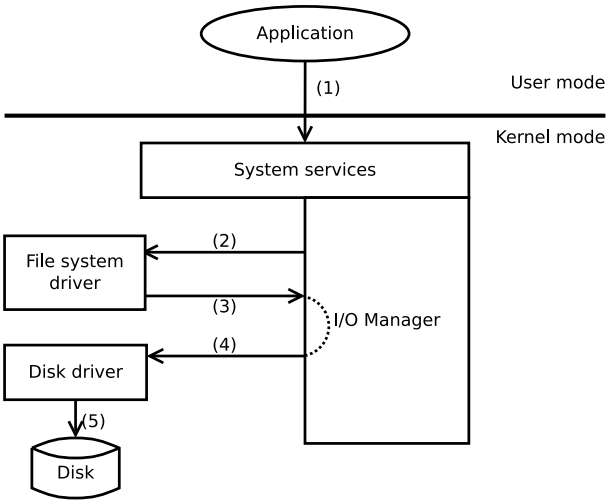

Figure 3.6.: Layering of a file system driver and a disk driver [21, pg. 536]
(1) NtWriteFile (file_handle, char_buffer)
(2) Write data at specified byte offset within a file
(3) Translate file-relative byte offset into disk relative byte offset and call next driver
(4) Call driver to write data at disk-relative byte offset
(5) Translate disk-relative byte offset into physical location and transfer data

3.5.4. Structure of a Driver

A driver presents the following entry functions to the operating system:

- init-function: is called, when loading a driver. This function fills the system structures, registers the rest of the driver and does global initialising work.
- add-device: if the driver supports PnP it needs to implement the add-device routine. The PnP Manager sends notifications to the driver via this routine, whenever a device for which the driver is responsible is detected. It creates a device object to represent the device.
- dispatch: the dispatch-functions are the main functions of the driver, these include open, close, read, write and others for representing the capabilities of the device, file system or network. The I/O Manager generates IRPs and calls these dispatch routines.
- start: initiate data transfer from and to the device. This function is only defined in drivers, which rely on the I/O Manager for IRP serialisation. The I/O Manager ensures that the driver only processes one IRP at a time.
- ISR: the kernel interrupt dispatcher transfers control to that function. It acknowledges the IRQ and queries requests for DPC-routines. This function is only present for interrupt driven devices.
- DPC routines: perform most of the IRQ servicing after acknowledging the IRQ in the ISR. Initiates the process of notifying the I/O Manager of a completed operation and starts the next queued operation on the device.
- completion routines: for layered drivers, this function is called by the lower drivers upon completion
- cancel: if I/O operations can be cancelled, the driver assigns a cancel routine to the IRP. If the thread cancels the operation (by calling the `CancelIO()`-function), the I/O Manager calls this cancel-function. It can release memory and complete the IRP with a cancelled state.

- unload: releases any system resources a driver is using, so the I/O Manager can remove the driver from memory. Unloading and loading a driver can happen at run-time.
- system shutdown notification routine: This function allows the driver to clean up on system shutdown.
- error logging: when unexpected errors occur (like a bad disk block) the driver's error logging routines note the occurrence and notify the I/O Manager. The I/O Manager writes that information into the error log file.

3.5.5. Plug and Play

The PnP Manager automatically recognises new hardware devices, both while boot-up and when the system runs. It is also responsible for resource allocation like interrupt lines, I/O memory, I/O registers or bus specific resources. It is able to do resource arbitration. That means it is able to reallocate resources to other devices, so every device meets its requirements for operation.

Loading a driver is also a concern of the PnP Manager, when a new device is attached. First the device has to be identified. If there is already a driver that manages this device installed into the system, it needs to be loaded. If not: the kernel-mode PnP Manager will communicate with the user-mode PnP Manager for installing an appropriate driver. Maybe the latter has to ask the user to locate a driver.

At last the PnP Manager implements mechanisms for drivers and applications to use to detect if a hardware device is added, removed or present in the system.

A device driver needs a PnP dispatch routine and an add-device routine. A bus-driver has to support different types of PnP requests than function or filter drivers do.

At system boot, the PnP Manager wants to enumerate all installed devices. So the bus driver of the primary bus is loaded and asked to enumerate all present devices. The devices are identified, their requirements determined, and the bus driver delivers information about all detected devices on the bus. The PnP Manager loads more bus drivers (for other detected buses) and function and filter drivers and calls their add-device routine for every recognised device.

The add-device routine will not communicate with the device, but it will do management preparations. Only when the PnP Manager sends a start-device command to the PnP dispatch routine of the driver, the driver is allowed to start communication with the device. The start-device command also tells the driver the resources usable by the device, which the PnP Manager had determined in arbitration stage.

After sending the start-device command, the PnP Manager can send other commands like device-removal or resource-reassignment commands. When a user wants to (safely) eject a device, the following steps happen internally:

1. the PnP Manager sends a query-remove notification to every application which registered for receiving PnP notification for that device
2. applications typically register on their handle, which they close on query-remove
3. if no application vetoes the request, the PnP Manager sends query-remove to the driver of the device

4. the driver can deny the removal, or wait for all pending I/O requests to be finished and deny new I/O requests aimed for that device

5. if no open handles remain and the driver agrees to remove the device a remove-request is sent to the driver

6. the driver will discontinue communication with the device and release all resources, which it had allocated for that device

The PnP Manager will send a surprise-remove command, when the user removed the device before (safely) ejecting or the device failed. It informs the drivers that they may no longer communicate with the device as it is no longer present.

The PnP Manager can also reassign resources at run-time. It sends a query-stop request to the driver. It can decide to either deny the request or complete pending I/O requests and queue new requests. So resource reassignment is transparent to the applications. The stop-command is issued in the latter case, the assignment is changed and the start-device command is sent to the driver to begin communication with the device once again.

There is a well-defined state-transition table for Windows drivers with their appropriate commands.

3.5.6. Power Manager

See Section 2.2.1 for the ACPI conventions. The power management is split between the Power Manager, which owns the system power management policy and the device drivers, which enact the driver policy. The power Manager decides which system power state is appropriate at any given point in time.

When a state change is required, the Power Manager instructs the device drivers of devices, which are capable of changing their power states, to perform a power-state transition. The Power Manager considers the system activity level, the battery level, user actions, requests from applications and the power settings in the control panel.

In the enumeration stage of PnP the drivers describe the capability of the devices to switch to power states D1 and D2 and optionally a latency. The bus drivers construct a table of system states and the lowest possible device power settings (most devices are turned off (D3), when leaving S0).

When switching the power state of the system, the drivers are notified by their power dispatch routine. They may request other drivers to bring their devices into a different power-state. The Power Manager ensures, that not too many power-state changes are done concurrently, as some device need a huge current to power up.

The Power Manager will ask every device driver if a power-state change is acceptable. If not (e.g. because a device performs an I/O operation or does time-critical interactions), the system will maintain its current power settings.

A driver can also set its device(s) into a sleep state itself, when it is idle. Detecting the inactivity can either be done by the driver itself or by Power Manager facilities. A driver can set a callback function in the power Manager to check whether a device is idle.

3.5.7. I/O Data Structures (ntddk.h)

The Windows Object Manager keeps kernel-mode objects to represent some data structures. This thesis does not go into detail about what a Windows object is or how the Object Manager works. But the objects have some features, which are interesting. They are named, are system resources, which can be shared by user processes, are protected by object based security and support synchronisation.

File object (type `FILE_OBJECT`) is a kernel-mode data structure for handles to files or devices. When a caller opens a file or a simple device the I/O Manager returns a handle to a file object. File objects are protected by Access Control List (ACL), the I/O Manager checks of the requested access mode is allows on the `open`-call. Every time a thread opens a file, a new file object is created with its own set of attributes (as for example the offset can be different from thread to thread). Threads need to synchronise their access to the shared resource 'file'.

The driver object contains dispatch routines, so the I/O Manager can call them when the driver is needed. The device object represents a logical or a physical device. It describes its characteristics, like alignment requirements of its buffers or the location of the IRP queue. When a thread opens a handle, the I/O Manager needs to be able to find the driver and device objects.

The I/O Manager creates the driver object, when it loads the driver. The PnP Manager create the device objects, when the add-device routine is called in Windows 2000 drivers. The I/O Manager unloads the driver, when the last device object is deleted and no references to it exist anymore.

Device objects can have a name in `\Device\` in the Object Manager namespace. The namespace `\Device` is inaccessible by user applications with the Win32 API. Legacy drivers use well-known names, whereas PnP drivers use the devices Globally Unique Identifier (GUID) (a unique device identifier) to name their objects. A symbolic link to the `\??` is used to allow Win32 applications to access the devices. The application can enumerate devices for a given GUID and query additional information.

The device object contains a pointer to the driver object. The I/O Manager knows which driver to call, when it receives an I/O request for that file. It uses function codes to signal the device driver which function it requests.

An IRP stores the file object and has normally a header with the type and size of the request as well as pointers to a user buffer for buffered I/O. The IRP then contains one or more stack locations, where the file object, the function code (major for identifying the driver dispatch routine and a minor as a modifier, e.g. used by Power Manager or PnP Manager) and function specific parameters reside.

Most drivers only implement a subset of the major function codes. Most commonly: `create` (open), `read`, `write`, `device I/O control`, `power`, `PnP`, `system` (for Windows Management Instrumentation (WMI) commands), `close`. The I/O Manager fills the missing functions with references to its own `InvalidDeviceRequest` routine.

IRPs are stored in a queue by thread, which allows the I/O Manager to cancel all outstanding requests, when the thread is terminated or terminates.

Driver Loading is possible at system boot (at different times) or due to the enumeration of devices by the PnP Manager. A driver can alter its boot order with values in the registry, it can be loaded by the

OS loader, by the I/O Manager or after executive subsystems have finished their initialisation routines.

Bus drivers and file system drivers are loaded by the bootloader. There are several later levels, including an on-demand level (which indicates loading by the PnP Manager, when the device is recognised later on).

The root driver represents the entire system and acts as bus driver for legacy drivers and the HAL. The HAL enumerates the devices in the motherboard, fans and batteries, but relies on the hardware description of the set-up process to detect the primary bus. The primary bus driver enumerates all its devices, maybe including additional buses. Additional drivers are loaded until all devices are detected.

The PnP Manager creates a device tree, which represents the relationship between devices. The nodes of the device tree, also called devnodes, contain the so called device stack, which is a stack of device drivers for the device, and additional information for the PnP Manager [15]. The device tree is used by both the PnP Manager and the Power Manager, when issuing commands.

Selection of a Function Driver for a Device The bus driver enumerates attached devices and reports IDs of the devices. E.g. for USB-devices the Vendor ID and the Product ID are used, together called device ID. The device ID and the instance ID (to distinguish several devices of the same product) are used to load the driver through a configuration in the registry.

I/O Processing Most I/O operations are synchronous, meaning, that the caller-threads wait in kernel-mode until the operation is finished. Asynchronous I/O has to be specified, the threads then have to synchronise themselves. It is also possible to map the contents of a file into process memory, so the process is able to read and write to it with normal memory operations. The Memory Manager uses the paging mechanism to load and write data to and from the disk. Also Scatter and Gather operations are possible, to read / write from / to several buffers to / from a single continuous file.

When the I/O operation is finished the device issues an interrupt and wants service. For asynchronous access an APC is necessary, which run in the context of an application, whereas DPC could run in any arbitrary context. The APC copies the necessary data into the virtual memory of the application and sets an event object or completion port to signal the state. It then frees the IRP memory. A user application can also supply an additional APC, which is called as last function, before the completion of the I/O operation.

3.6. JX

JX is a research operating system by Golm, which uses the type-safe instruction set of the Java virtual machine to implement an operating system [8]. ‘In a type-safe instruction set instructions (operations) are applied to typed operands that refer to typed data entities. Strict rules describe what types of operands can refer to what types of data entities and what operations can be applied to them’ [8, p. 1].

JX incorporates a clean room implementation of a Java bytecode interpreter, which translates Java bytecode statements at run-time to x86-instructions. The microkernel is the only code, which is written in an unsafe language (C and Assembler), many of the system processes are implemented in Java bytecode. JX uses interrupt service routines in Java.

3.6.1. Architectural Overview

JX uses the type safety property of the Java bytecode as its sole method of security. It runs otherwise in a single address space and does never leave the supervisor mode of the processor. The microkernel provides functionalities for other parts of the system, like:

- system initialisation
- saving and restoring the CPU state
- inter-Domain communication
- low level support for protection Domains and the component management
- garbage collection

The microkernel is split into two parts, the run-time system for Java bytecode and a number of system services, which can be used by other parts of the system. A Domain is the unit of protection in JX. Every Domain can be seen as its own separate JVM, so Domains are isolated from each other. Portals are the sole mechanism for communication between Domains, which is used similarly to Remote Method Invocation (RMI) or Remote Procedure Call (RPC).

The Microkernel is designed as an exokernel. The design of the kernel focused on three design principles:

- avoidance of globally shared state
- avoidance of dynamic resource management
- move as functionality to the level of the type-safe instruction set as possible

State is shared on Domain level, not in the microkernel, there are no dynamically growing data structures in the kernel. All per-Domain structures are kept in the memory of the Domain. This is necessary as the microkernel is the only part of the system, which cannot be replaced at run-time. It is safe to place data structures on the Domain's heap, because it cannot access its own memory directly due to the type-safe nature of the Java bytecode.

JX seeks to keep abstractions to files on the Domain level. Also the microkernel keeps some spots open, like the CPU scheduling, ISRs, locks or condition variables, which call certain Java code pieces. This reduces the number of transitions between the Java and the C level. The kernel itself does not use mutexes or semaphores, but saves its critical sections by disabling interrupts or with spinlocks.

Also the JX microkernel tries not to provide virtual resources and the support for the physical resources is held at a minimum. The kernel only manages memory, even the CPU may be delegated to Java level.

3.6.2. Processes

A domain is the unit of protection in JX, meaning that a 'domain contains a heap, a number of code components, threads, portals and services' [8, pg. 27]. The heap of a domain is automatically managed by a garbage collector. The implemented garbage collectors become active when a thread allocates memory, but the heap does not have enough free space to fulfil the allocation request. Threads cannot migrate between domains, not even when communicating.

DomainZero contains the microkernel itself and services for other domains, like creation of domains. After the creation of a new domain an entry method is called, which can accept an array of objects and an array of strings, this can also include portal objects. A domain can only access its own data and methods or invoke methods exported by other domains through portals. Faults of one domain do not propagate to other domains and granted resources can be used by a domain without affecting any other domains. DomainZero is the only domain, which cannot be terminated.

A domain can interact with DomainZero by either explicitly invoking services of the DomainZero, like the naming service or by implicitly calling the run-time system, like checking a downcast or allocating memory for a new object. Domains may share code (classes, interfaces) with other domains, but they all have their own set of static fields.

'The portal invocation is the central inter-domain communication mechanism' [8, pg. 35]. JX' portal invocation was designed to fit into the object oriented programming model and to be similar to RPC and RMI. Portals are the sole method for communication between domains. A portal object is the representation of a service of another domain. A portal invocation behaves similar to a normal function or method call. The calling thread is blocked, the service thread executes the method, returns the result and can then serve other requests. The calling thread is unblocked when the service method returns.

Portal communication is implemented as message passing between domains. Buffering of messages is the responsibility of the source domain, synchronising accesses, flow control, and message acceptance of the target domain. DomainZero exports some functionality as so called fast portals. The fast portal invocation is executed in the calling thread, rather than a service thread. There is no switch to the service thread necessary. For some functionalities switching to a different domain would make no sense, e.g. for allowing a thread to yield the processor.

A portal is a proxy of an object in another domain, which can be accessed by RMI. The entity, which is accessed by another domain is called a service. A service is comprised of the implementation of the portal interface, a service thread and the initial portal. A portal can be thought of as capability, which can be copied between domains. A portal is typed, other than pipes in Unix.

Domains

The heap of a domain is used to store all information about the domain including Thread Control Blocks (TCBs), stacks, portals, services or components. The heap is garbage collected. This thesis will not go into detail about garbage collection algorithms, though Golm [8] describes several algorithms implemented in JX.

Creating a domain is done via a method call to DomainZero. A code component is loaded into the domain or shared with other domains, a starting thread is created, which starts executing the entry method, which was specified on creation. The entry method is a static method of the loaded component. It takes an array of strings and an array of objects as parameters, including portals to services, like the naming service. DomainZero also exports a method for returning the naming service's portal.

Terminating a domain can be done at any point in time. A security service decides whether a domain may be terminated by a specific other domain. A domain is automatically terminated, when it does not have any running threads, no services, no references to thread from the outside (which may be used to unblock a thread from the outside) and no registered interrupt handlers. If all four conditions are true, the domain does not have any internal activity and cannot be activated from the

outside.

When a domain is terminated, all threads are stopped, all services deactivated and active service calls terminated by raising an exception to the caller. A thread waiting for the completion of a portal call is stopped and a signal is sent to the portal, that the calling domain is terminated. The called domain may decide to stop the activity immediately or at any later save point. The reference counter of all referenced remote objects by local portals are decremented and the fixed memory areas and the heap are released.

Isolation domains are isolated in regards to resource usage and data access from other domains. A domain may only access its own data structures or methods and exported methods of other domains. A domain may fail due to a software bug or hardware fault - the domain can then be restarted or removed without affecting the rest of the system. All resources like ISRs and memory must then be freed.

Resource management Once a resource has been committed to a domain it can be used freely, for example heap memory can be used to allocate new objects with an arbitrary memory management scheme. JX divides its resources in three categories:

- Primary resources - memory and CPU-time; necessary for computations
- Additional or secondary resources - disk, network, display or input devices; require primary resources to be used
- Virtual resources - built upon primary and secondary resources; virtualisations of primary and secondary resources, which allow concurrent, easy and portable access using a high level API.

When a domain obtains a portal to a service it obtains a resource. When it is deleted and the reference count hits 0, the `ServiceFinalizer` is called, for clean-up of resources allocated for that specific domain.

Execution environment A domain provides an execution environment to Components. This includes the memory management by identifying the Garbage Collection (GC) strategy if any, scheduling and synchronisation by using specific thread management strategies, including mutex and condition variable implementations and a scheduling strategy. Also a domain can configure, how incoming service requests are handled and whether a thread pool should be created for incoming requests, when a service is initialised.

A domain also provides a namespace for the components, as a domain may only access a name service portal at first. The name service portal is provided by DomainZero for the first domain. When a domain creates a new domain it may use a different name service than the default one for the new domain.

3.6.3. Interprocess Communication

The central interprocess (interdomain) communication mechanism is the Service/Portal. Also they only depend for an exact amount of time on each other. Two mutual distrusting domains can communicate with each other, as with services a denial-of-service attack and obtaining access to data structures without the permission of the owner is not possible. The abstraction is already known to Java developers, as the mechanism is designed after the Remote Method Invocation (RMI) model. It

is also possible to extend the model to include communication with remote machines.

The local object, which is accessible by other domains is called a Service. A portal is the remote reference to a service object, which may also be passed to other domains. A portal type is an interface derived from the marker[3]-interface `jx.zero.Portal`. Portal calls behave like normal synchronous method calls, i.e. the calling thread is blocked, the service thread executed the method-code and returns a value, then the calling block is unblocked and the service thread is ready of another request.

The service needs to provide the portal interface as well as the implementation and its instantiation (service object), the client can then call methods according to the interface. The code using a portal may not make any assumptions on whether the service object is actually local (in the same domain) or remote (in another domain) for portability's sake. When a thread blocks, every object has to be in a consistent state, every client has to consider that before making a portal call, may it block or not.

Parameter passing Calling a service method of another domain has the problem, that the parameters must be accessible by the domain of the service. One possibility could be to have shared objects or a shared heap, but that makes CPU and memory accounting very difficult and may make programs more complex as a shared object needs to be created explicitly on a shared heap. Also a shared heap makes fault containment very difficult, and may result in one domain failing because another failed before (and corrupted the shared heap).

JX copies the parameters, which has a performance impact especially with large objects or object graphs. Although most programs only needs small objects as parameters to service calls. Also the object identity is lost on copying to the service domain. The object graph can be copied by the caller or the called domain. However when the caller copies the data before being blocked, it wastes heap space of the target, as the whole reachable object graph is copied. With very large object graphs a denial-of-service is possible, therefore the amount of copied data is limited in JX.

Another problem is identifying cycles and stopping the copying (of a cycle), when reaching an object, which is already present in the called domain. One solution is to use a mapping table ($src : dest$) to look-up whether the source object was already copied and if so use the destination reference instead of copying again. When the garbage collector runs on the target heap while copying is in process, the copying is aborted and restarted when the GC is finished. No other thread may access the source heap while copying, so the garbage collector cannot run there, while parameter copying takes place.

However the source and destination graphs may not be identical especially with these objects:

- **portal** - create a correlating portal in the target domain
- **portal, which returns home** - a portal, which references a service in the target domain may be substituted by a reference to the service object
- **service (interface class)** - promoted to service and portal, i.e. the service is started and the portal to the service is copied.

There are three problematic situations, as depicted in Figure 3.7:

(a) references between service graphs

[3] A marker interface in Java presents the JVM with an attribute for the class, i.e. an object of a class may be `Serializeable` or `Cloneable`, without having any attributes or methods attached to the interface itself

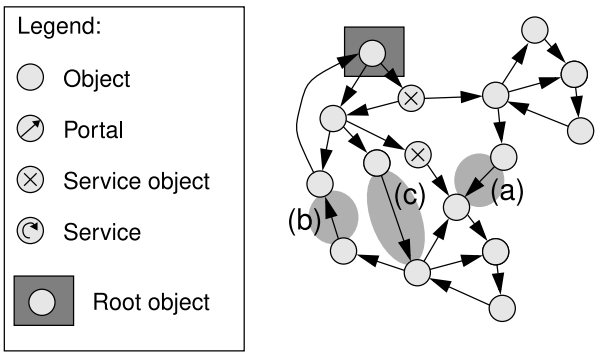

Figure 3.7.: Problematic connections in an object graph used as parameter [8]

(b) references from a service graph to a parameter graph

(c) references from a parameter graph to a service graph

Conditions b and c can lead to inconsistencies and parts of a service graph are copied to the target domain. These should therefore be avoided.

Implicit portal parameters It's possible in JX to attach an object to the current thread, which is passed over portals implicitly. This feature can be used for credentials for example and makes the code more readable. These implicit parameters are passed through the invocation path, i.e. when the service needs to access another portal.

3.6.4. Device Drivers

Memory Objects are available in three different flavours in JX. They can be useful for accessing memory mapped I/O registers.

- memory object
- read only memory
- device memory

The memory object and the read only object need both a size as parameter for their creation, whereas the device memory needs an address and a size. JX implements a simple mapper for Java objects to memory objects (see struct pointer in C). The mapper can only map simple integer types.

A problem arises with DMA and the GC of JX. It must be ensured, that buffers, which are currently transferred may never be freed and potentially overwritten while the transmission is still going on. The programmer could keep the object alive by managing a list of active DMA transactions and therefore keeping a reference of that buffer so it will not be garbage collected.

Device Drivers in JX are written as untrusted part of the operating system, i.e. written in Java. A driver runs in a domain, like a normal application and may emulate the functionalities of a device. A real driver needs to do the following:

- handle interrupts
- access physical memory at specific locations

- access device registers
- control DMA transactions

Interrupt Handling Interrupts are handled by dedicated IRQ handler threads, which are created as one-shot threads by the system, when the IRQ occurs. The Interrupt Manager of DomainZero allows for registering an object which contains the IRQ handler (`handleInterrupt()`-method) and has a method `installHandler()`, which installs the IRQ handler for the calling CPU on a multiprocessor system.

The `handleInterrupt()`-method handles an interrupt, with other interrupts disabled on the interrupted CPU. The method is the first level interrupt handler in JX, as it is called when the interrupt occurs, acknowledges it and unblocks the thread that handles that interrupt asynchronously (similar to bottom half of Linux or DPCs in Windows). `handleInterrupt()` should only run for a set amount of time, because it cannot be interrupted, so it should only do the bare-minimum for acknowledging the interrupt and deferring the work of handling the device-state to a different thread, which does not have a higher IRQ level.

The unblocked thread for handling the device state after an interrupt has been acknowledged is scheduled by the regular thread scheduler of the system and does not have a higher IRQL, i.e. it can be pre-empted at any time. JX does not allow to prioritise ISRs, i.e. an ISR cannot be pre-empted, as that would introduce more problems and the ISRs are very short. It would become necessary to share data between code of different levels and add an additional scheduling policy. Pre-empting ISRs would become necessary with very slow devices or devices that would need complicated computations inside the first level interrupt handler.

A first level interrupt handler should not block the system indefinitely. The bytecode verifier of JX analyses the classes that implement the `InterruptHandler`-interface. The `handleInterrupt()`-method may only take a certain amount of time. To allow checking and avoid undecidable situations only a simple code structure is allowed. Only linear code, loops with constant bounds and no write access to the loop variable inside the loop is allowed.

Device Registers can either be mapped to a specific memory location or accessible by specific instructions. If they are mapped to a memory region, the `DeviceMemory`-object can be used to access these registers. Java does not allow direct access to instructions like `in` or `out` by Java-applications. JX implements these instructions as fast portals in DomainZero, i.e. as code, which can be inlined by the run-time-translator.

Garbage Collection Drivers run in a normal JX domain, i.e. driver will also be subject to garbage collection. 'At the moment' JX does not support running threads, while the garbage collection runs. When an interrupt occurs, while garbage collection is ongoing, the system will defer handling that interrupt until the GC is finished. Golm proposes two solutions to that problem:

1. drivers may only use incremental Garbage Collection (so that a mutator thread can run in parallel)
2. drivers may not allocate dynamic memory

Normal Java programs allocate new objects very commonly. This behaviour can be checked by the verifier before running a method. A way of handling dynamic memory allocation is, that a driver is allowed to allocate memory until a certain point X, but not afterwards. This restricts the driver in

several ways. First new threads can only be created until that point X, as the thread control block and the stack need to be created in the heap. If the driver needs several threads it has to create them beforehand and unblock / block them as needed.

Portal invocation, i.e. calling functions of other domains, copies data between domains, which will be allocated in the heap. After the point in time X it will not be possible to use portals if the parameters or the return value are of object type.

Lastly throwing exceptions may be a problem. Exceptions could be thrown using a pre-allocated object, but these could not capture the current state of execution (like a stack-trace), which would limit the use of exceptions to a debug program. But the control will go to the exception handler function, which is fine for most of the cases [8]. The JX microkernel can be configured to use pre-allocated objects for run-time-exceptions as well.

Direct Memory Access JX is built from the standpoint, that the device manufacturer can be trusted, but not the driver provider, i.e. the device will operate as specified, but the driver may behave in a strange way. DMA enables a device to bypass the protection mechanisms of JX by allowing transfer of data to different memory locations. The code for controlling DMA needs to be trusted. The driver is therefore split into a trusted simple part and a complex untrusted part.

Synchronisation Drivers are not allowed to disable interrupts outside the interrupt handler. Usually disabling interrupts is a cheap way to avoid race conditions. Locks are also not allowed in an ISR, because ISRs may never block. JX therefore implements `AtomicVariable` with a value, which can be set or gotten. Also the atomic variable can atomically check and set with a parameter or block if the stored value is equal to a parameter. In the latter case another thread needs to set the value of the variable and unblock a blocked thread. The methods disable interrupts on uniprocessor systems and use spinlocks on multiprocessor systems.

The Device Driver Framework is not part of the core JX operating system, but serves as an example for a driver framework on top of JX. For device identification every driver needs to provide an implementation of `jx.devices.DeviceFinder`. The method `find()` returns an array of supported devices of that driver. The class `jx.devices.Device` is a supertype of every device. It has the following methods:

- `open` and `close` to (de)initialize the device
- `getSupportedConfigurations()` returns an array of device-specific configurations, which can be used as a template

`open` needs a fixed configuration object to specify how the device will be used. JX Device Driver Framework defines a set of categories such as network disk or framebuffer-devices, which sets an interface for each categories, a driver has to implement. That interface will be used for communication between layers of drivers. The interface also has to be implemented if a driver emulates a device or provides additional features based on the services of a device or driver.

3.7. JavaOS

JavaOS is an operating system developed around 1997 to 1999 by JavaSoft and IBM. It is very interesting to look into JavaOS, as it tries to be a minimalist operating system, so to implement only the exact level of support required for the JDK, as well as implement as many functionalities in Java as possible.

JavaOS is comprised of a small microkernel written in C, which contains the thread scheduler as well as the 'bulk of the JVM, including the garbage collector, interpreter, and other run-time facilities' [20]. Below that there is a hardware abstraction layer, which implements traps and context switching. The whole system resides in a single address space and requires no memory management unit.

Ritchie [20] points out six advantages of using Java instead of C and Assembler for systems programming:

- flexible object model
- portability
- good performance
- automatic garbage collection
- dynamic extensibility
- rich debugging and development tools

Flexible object model Java uses the object oriented model to create software. Services such as drivers and event processing seem to be natural candidates for object decomposition. Ritchie states, that a generic NetworkDriver-class could present the method calls `ReadPackage` and `WritePackage`, which every network driver would have to implement when inheriting the abstract class. This would not stop a concrete driver to implement additional functions, which other layers in the network stack could use, after casting the generic NetworkDriver-object to the more precise type. This would allow to have good names for method calls, instead of the `ioctl()`-function of Unix-systems, which implements operations not covered by the common set `read`, `write`, `open`, `close`.

The event processing would also benefit from object orientation. Events can be produced by external devices or the device name space tree as a synchronous or asynchronous event. The most basic form of an event is an `OSEvent`, which is everything the event subsystem needs to manage.

Portability Java bytecode in itself can be run on any platform, provided the correct virtual machine runs the code, i.e. the virtual machine runs on the target hardware. However 'systems programmers often massage bits on processors and devices whose architecture models vary widely' [20]. Most of the code, is platform independent and runs even without re-compiling on new hardware, but some drivers and more chip-specific code would have to be re-written or replaced.

Performance The used JVM runs Java code faster than on a conventional system. Ritchie mentions, that the functionality of the Java VM or the JDK would need mapping to the operating system's services, which in turn requires glue code, maybe even data conversion. This is not the case on a pure Java operating system, even without JIT-compilation.

Another reason for good performance of JavaOS could be, that it does not need to alter any paging tables, as everything is contained within a single address space, i.e. context switching does not need a Translation Lookaside Buffer (TLB)-flush. Also if a thread calls on a TCP write, the same thread may go down the protocol stack and execute the driver code itself, i.e. writing the package itself to the device. Security in JavaOS is handled by the programming language and mostly load-time or run-time checks in software, so the system can prevent processes from modifying each other's memory.

Garbage collection Automatic GC prevents threads from prematurely freeing data, which is in use by other threads, or not freeing memory after usage. However there are some cases, in which the garbage collector can ruin the performance. For the network stack a number of buffers could be allocated frequently, which the garbage collector would need to collect very frequently. Instead

JavaOS keeps a list of free buffers, and returns the buffers to the free-list after usage to prevent the garbage collector from collecting the buffers.

Extensibility Ritchie [20] points out, that a conventional operating system can load modules at run-time but often times fails to unload them, when they are no longer in use. In JavaOS classes can be loaded at run-time, as since the JVM views them identical to object, the GC can collect them, when they are no longer in use.

Debugging and development tools Identifying the state of a raw processor is very difficult. It is possible to use protocols like JTAG to view the memory or a system or list the register values, but stopping a java virtual machine and changing bits is easier. An interesting thing to note is, that Ritchie et al. 'spent more time debugging [the C-based virtual machine], than all the other parts of the system combined' [20, pg. 35].

3.7.1. Device drivers in Java

JavaOS comes with two main APIs for constructing device drivers: the `Interrupt` class, which helps manage interrupt handlers and the `Memory` class, for accessing memory mapped device- and bus registers. The `Interrupt`-class provides register and unregister-functionalities for Java interrupt handlers. The `register`-method registers an `InterruptHandler`-object with the machine level interrupt vector. When the hardware sends the specified signal, the `interrupt`-method of the registered object is executed, which handles the device interrupt accordingly, for example by checking the device's state register.

The `Memory class` provides primitive access to memory mapped device registers. It allows reading and writing with sizes of 8, 16, 32 and 64 bits, as well as bulk array copies. Some architectures however use I/O ports instead of memory mapped device registers, so the programmer would still need to get down to assembler-level to use the appropriate instructions (like `inp, outp` on x86).

3.8. Fuchsia

Fuchsia is a relatively new operating system from Google. It was announced in 2016 and a prototype version of the system has been released and the major concepts of the system have been documented [9]. Unfortunately main aspects of the operating system design have been left out, like the boot-up, processes or details of the Interprocess Communication (IPC) facilities. Some critical concepts however have been documented deeply, like the driver and device management or design decisions regarding POSIXy interfaces.

The Zircon microkernel implements the Device Manager, core and first party device drivers, the LibC and launchpad, a library for loading and using Executable and Linking Format (ELF)-binaries. Fuchsia also defines an Interface Description Language (IDL) to describe interfaces between processes.

3.8.1. System Calls and LibC

Fuchsia does not adhere to the POSIX standard system call API. Syscalls in Fuchsia use a concept called Kernel Objects, which are represented to user space by a 32-bit integer, a so called handle. When calling the kernel to manipulate or use one of these kernel objects, the handle has to be provided. The kernel checks that the handle exists, its type and its rights.

There are three categories of syscalls in the Zircon kernel:

- system calls, where no handle is provided and no limitations exist, which rights are needed to use the system call. E.g. `zx_clock_get`, returning the current time.

- system calls, where a handle is provided as the first parameter, denoting the object to act upon. E.g. `zx_channel_write`, writing data to a channel object.

- system calls, which create kernel objects, but don't take handles to other ones. E.g. `zx_channel_create`, creating a new channel object.

At any given time, there are one or more handles per object in one or more processes. If the last handle to an object is removed, the object is destroyed or put into a final state. Handles can be moved from one process to the next one by writing them into a channel-object, or by passing it on process creation. Two handles to the same object may exist in the same process, as they may have different rights attached to them. Zircon knows three syscalls, which take care of handles (from [9]):

- `handle_close` - closes a handle

- `handle_duplicate` - creates a duplicate handle (optionally with reduced rights)

- `handle_replace` - creates a new handle (optionally with reduced rights) and destroy the old one

Kernel Objects have an identifier, a 64-bit integer, which is unique for the run-time of the system, they are never re-used. The kernel also has an identifier and a Null-Identifier is defined, to mark an invalid-number.

An object may have so called signals (not to be confused with POSIX-signals), which indicate certain states of the object. For example a socket or a channel may be readable or writable. A thread may choose to wait for a single or several signal(s) to become active on an object. It may also create a so called port, which is an object, other objects are bound to. The thread receives packets with information about the pending signals. An event is the most simple object, with no other state, than the collection its signals.

Fuchsia brings a LibC implementation, which is based on the musl-LibC. The C-subroutine library implements the C11 standard, which also includes threads and mutexes. It requires some glue code, to map the C11 structures, like `thrd_t` to `zx_handle_t`. Programs may bring their own C-library if they need additional functionality not provided by system LibC. Fuchsia ships with a C subroutine library, programs may use.

The POSIX API is implemented in parts in the LibC, but large parts are omitted, as they don't fit the syscall interface of the Zircon kernel. For example `fork`, `exec` or the signal-concept (see Section 3.3.2) don't exist.

As Zircon is a microkernel it is not seen as its job to care about file I/O, so calls to `open`, `read`, `write`, ... simply fail. Fuchsia uses a second library, *libfdio.so*, which implements communication to user level parts of the Fuchsia OS, which implement file I/O and provide a POSIXy interface layer. The same is true for the sockets library: the network stack is implemented in user level. POSIX threads are implemented only in core aspects, which map straightforward onto C11, but other parts are not provided. The implementation does not aim to be comprehensive.

3.8.2. Process Model

Fuchsia knows threads, processes and jobs. A thread is the unit of execution and has its own CPU state and stack. Threads exists inside a process (address-space), which in turn is owned by a job. A job defines various resource limitations, has parents, which form a tree, to the root-job. A thread cannot create a new process or job, if it does not have a job handle.

Fuchsia uses message passing as IPC. It facilitates the concepts of sockets and channels, which are bi-directional and two-ended. Creating sockets or channels returns two handles, one per end-point.

A socket is stream-oriented, i.e. data may be written to or read from it in units of ≤ 1 Bytes. Sockets may force short writes or reads, if not enough data is read or the buffer is full, so the requested size-requirement of the read or write-operation cannot be fulfilled. Channels are datagram-oriented, there is a maximal message size. Either the message fits or not, short read or write-operations are not supported by channels.

A special case is reached, when handles are transported over channels or sockets. When a handle is written into a channel, the handle is removed from the sending process, when it is read, it is added to the receiving process. The handle continues to exist inside the channel, so the object is not deleted, except when the read-end of the channel or socket is closed.

3.8.3. Device Driver Model

Zircon device drivers are ELF shared libraries, which are loaded into Device Host (devhost) processes. The Device Manager contains the Device Coordinator, keeps track of drivers and devices and manages the discovery of drivers. It also cares for the creation and direction of devhost-processes and it maintains the device file system (devFS). User space applications use the devFS to access devices.

The Device Coordinator views devices as a simple unified tree. Branches of the tree indicate that the device drivers are loaded into the same devhost process. Drivers can be loaded into the same devhost process for improved performance or split up into several ones for security and stability reasons.

Drivers may implement so called protocols, which are RPC-like interfaces, for using standard functionality of the same class of devices easier and more convenient. A class promises, that the driver for a device implements a specific protocol. Protocol calls can be used by user space applications, but also by other device drivers.

When a device driver is needed it is loaded into a devhost process. The so called Binding Program, which is a description of the driver inside the binary, determines which devices the driver can serve. Listing 3.1 shows the source code representation of an example binding program of a driver for an Ethernet device.

Listing 3.1: Binding program of an Ethernet device driver

```
1  ZIRCON_DRIVER_BEGIN(intel_ethernet, intel_ethernet_driver_ops, "zircon", "0.1", 9)
2      BI_ABORT_IF(NE, BIND_PROTOCOL, ZX_PROTOCOL_PCI),
3      BI_ABORT_IF(NE, BIND_PCI_VID, 0x8086),
4      BI_MATCH_IF(EQ, BIND_PCI_DID, 0x100E), // Qemu
5      BI_MATCH_IF(EQ, BIND_PCI_DID, 0x15A3), // Broadwell
6      BI_MATCH_IF(EQ, BIND_PCI_DID, 0x1570), // Skylake
7      BI_MATCH_IF(EQ, BIND_PCI_DID, 0x1533), // I210 standalone
8      BI_MATCH_IF(EQ, BIND_PCI_DID, 0x15b7), // Skull Canyon NUC
9      BI_MATCH_IF(EQ, BIND_PCI_DID, 0x15b8), // I219
10     BI_MATCH_IF(EQ, BIND_PCI_DID, 0x15d8), // Kaby Lake NUC
```

```
11 ZIRCON_DRIVER_END(intel_ethernet)
```

The macros will force the compiler to place the binding program into a note section of the ELF-binary. The Device Coordinator can then inspect the binding program without loading the driver. A driver has the following four methods:

- `init` is invoked, when the driver is loaded. It does the global initialisation, when they are needed. Typically none are necessary. If the method fails, the system regards as the driver load to have failed.
- `bind` is used for offering the driver a device to manage, according to the binding program. If the method succeeds, the driver *must* create a new device, which will become a child device of the one passed as argument to the `bind`-method.
- `create` is only called on platform / system bus drivers or proxy drivers, so not necessary for the vast majority of drivers.
- `release` is invoked, when the driver is no longer needed before unloading the driver. This occurs, when the last bound device of that driver is destroyed.

The device-protocol Device drivers implement a set of methods, which can be used on a device. The following hooks are specified for a generic device:

- `open` - calls, when the devFS-file is opened or when an existing instance is closed. May be used to disallow simultaneous access or return a new instance of the device. The parameter `dev_out` is used to return a new device instance.
  Default: return `ZX_OK` Default: return `ZX_ERR_NOT_SUPPORTED`
- `close` - called, when the connection to the device is closed
  Default: return `ZX_OK`
- `unbind` - called, when the parent is removed (due to hot-plug or fatal errors). It can be assumed, that no more `open`-calls will come, but I/O operations may continue to come. The driver adjusts the state of the device, brings clients to closing the connection and continues to answer all device-hooks until `release` is invoked. Calls `device_remove` on the device itself, when ready.
- `release` - after `device_remove` was called, all children are unbound, removed, connections closed and the device itself removed. Used for freeing associated memory. After returning, no access to the `zx_device_t` structure is allowed and no hook calls should be answered.
- `read` - non-blocking read-operation.
- `write` - non-block write-operation. Default: return `ZX_ERR_NOT_SUPPORTED`
- `get_size` - if the device is seekable, return the size of the device, i.e. the offset at which no more reads or writes are possible.
  Default: return 0
- `ioctl` - device specific operations, must not block.
  Success: return `ZX_OK`
  Default: return `ZX_ERR_NOT_SUPPORTED`

Inside a devhost process, the hierarchy of devices is represented as a tree of `zx_device_t`-structures. A driver may call `device_add` to add a new device as the child of either the device passed to their `bind`-method, or by a device, which was created by the same driver. The newly created device is then added to the devFS.

Device-structures are reference counted. A call to `device_add` increases the reference count, `device_remove` decreases it. When it hits zero, the device is unbound. If a parent is unbound, the `unbound`-method of the device is called. The device itself is then shutdown, and its children are unbound.

When the parent device is unbound, the child may still work. The parent device-methods are advised to return errors on new calls, as the child should be prohibited, to start a lengthy communication or new work for the parent device. When the devices are unbound, the `release`-method is used to remove any data structures associated with that particular device. After it has returned, it's illegal to access the devices `zx_device_t` structure.

3.9. Conclusions / Comparison

This Section compares the facilities of the discussed operating systems in terms of process management and IPC as well as their driver management.

3.9.1. Process Model and IPC

The process model and IPC infrastructure was described for several of the operating systems, like JX, 4.3BSD and Fuchsia. The three discussed OSs feature wildly different approaches to communication between their concepts of processes. Table 3.1 gives a quick overview.

Table 3.1.: Process models and IPC of JX, 4.3BSD and Fuchsia

	4.3BSD	JX	Fuchsia
Unit of Protection	Process	Domain	Process
Unit of Scheduling	Thread	Thread	Thread
IPC Mechanism	Pipe, Socket, Signal	Portal / Service	Channel, Socket

4.3BSD uses Berkeley-sockets of communication between processes and fits the pipe-functionality onto the socket-library. Sockets allow two processes to communicate to each other, even if they are not in direct relationship, whereas pipe allows parent-child-processes or two child-processes to communicate.

JX facilitates an RMI-approach, as it defines so called service-objects, which export methods to other domains. Domains have an own heap and cannot share any state with other domains in the system. The service communication model allows two mutually distrusting domains to communicate with each other, as the data provided by the domains is typed in communication. Service objects live only in one domain, whereas every other domain may have a portal (stub) to that service object. The portal has the same methods, but will require communication with the service object to solve its tasks. The calling thread is blocked.

Fuchsia knows two concepts for interprocess communication, the channel and the socket. A socket can be used as a byte-stream and allows short writes and reads, whereas the channel is datagram

oriented and does not allow short reads or writes. So called handles to kernel objects play a huge role in Fuchsias design.

3.9.2. Driver Model

The driver architectures of different operating systems were discussed earlier, as well newlib system calls, which need implementing on bare metal hardware.

Comparing the system calls of newlib with the file operations of Linux 2.6 unveils, that the C subroutine library is intended to be used with files as abstraction. Many calls are similar, like `close`, `fstat`,`lseek`, `open`, `read` and `write`. The Linux 2.6 file operations include `llseek`, `read`, (`aio_read`,`aio_write`, `readv`, `writev`) `write`, `ioctl`, `mmap`, `open`, `flush`, `fsync`. `readv` and `writev` are used as scatter / gather operations, `aio_read` and `aio_write` for asynchronous I/O.

The reason newlib needs a system call for these function is, that it cannot generically define the functionality of these calls, it cannot assume anything about the file system, to which it needs to write, when a `write`-call is issued. Maybe it does not have to write to a file system after all, but to `stdout`. Linux 2.6 uses the file operations table for the same reason. It cannot define the internal workings of these calls, and has to rely on different drivers, maybe different file systems, to implement these calls for their devices or files.

The described operating systems Windows 2000, 4.3BSD, Linux 2.6 do not differ greatly how they handle devices. Linux 2.6 abstracts devices as device files, which can be used by user-applications, just like 4.3BSD. Windows 2000 encapsulates devices in device objects, to which a symbolic link can be placed to a location where user-applications can access them.

The architecture especially between Windows 2000 and Linux 2.6 are different. Windows works with the concept of messages (IRPs), whereas Linux applications work with files, and invoke callback functions, which where registered upon the `open`-call of the (device) file. Initiating an operation on a device in Windows leads to the creation of an IRP, which is send to the corresponding driver by the I/O Manager. The driver may choose to create new ones and send them to other drivers.

Enumerating devices is also different. In Windows enumerating devices is done by the PnP Manager, which instructs the drivers to list all devices, whereas it is less organised in Linux 2.6. 4.3BSD does not have device enumeration after boot-up. Detecting devices is based completely on building the correct drivers into the system on compile-time and probing for the devices at known addresses at boot-up. Devices, which are added after boot-up cannot be detected.

Interrupt handling is the most similar thing in the three operating systems. In Windows the PnP Manager decides which interrupt lines are associated with certain devices, considering the requirements for all devices, whereas in Linux the drivers register their interrupt service routine directly to the line. Interrupt handlers are also registered in Windows to the line by the means of software lists of interrupt handlers to be executed when a certain interrupt occurs.

Then there are operating systems, which are a different by their design. Not only JX and JavaOS, which are written in Java, but especially Fuchsia, which is a conventional OS, i.e. written in C, but it does not feature the POSIX-system calls. Devices in the latter are represented by a tree of device-structures, which have hooks to be called, when an operation is requested for the device. The operations possible are the known calls `open`, `read`, `write`, `close` or `ioctl`, but with additional calls for managing

certain operations for hot-plug or removing devices at run-time. Drivers are loaded into the system in processes, when they are needed. Whether they are needed can be determined by reading the so called Binding Program, a special field of the ELF-binary of drivers.

JavaOS supports its drivers via memory and interrupt-abstractions and helper functions. A driver may choose to write an ISR in Java, and register it to a certain interrupt vector. The memory abstraction allows reading and writing memory values at exact addresses, which is used for accessing memory mapped I/O registers. However a C-like structure cannot be used, as the Java programming language does not support pad-bytes, notation of sizes of fields and may choose to re-order the field of a class. Also I/O ports are not supported by JavaOS, so the developer would need to write Assembler-code.

In JX interrupts are handled by one-shot threads, which are created, when an interrupt occurs. DomainZero allows for registering interrupt-objects to certain interrupt vectors. These interrupt objects contain a method, which runs as interrupt handler. A first level interrupt handler may not block, which can be analysed statically by the JX bytecode analyser. A problem is the garbage collector, as it may block an interrupt handler for an amount of time, if not taken care of. A JX-interrupt handler runs in Domain-context, so when garbage collection is ongoing, the execution of the interrupt handler is deferred until the garbage collector is finished.

Restricting interrupt handlers to certain functionality may solve the problem of a garbage collector run, when the ISR is running. Instructions for I/O ports are exported by DomainZero as fast portals. The access of memory addresses works conceptually identical to JavaOS.

4. Architecture of the interpreted Operating System PylotOS

In this chapter the general architecture of PylotOS is described as well as operating system design decisions. After discussing the machine model, the OS is built from the ground up and major concepts are explained. This includes processes, syscalls and Interprocess Communication. Then the device driver model is explained. System boot-up is briefly discussed, as well as a proposed extension of the interpreter to allow access to the internals of the virtual machine and to the real machine from inside the virtual machine.

Memory management and scheduling processes are not discussed in detail, as memory management is integrated in the interpreter already and scheduling policies do not change when implementing them in Python. Dispatching of processes however does.

The goals of PylotOS is to implement an operating system in Python without modifying the basic logic of the interpreter and implementing as little functionality in the interpreter itself as possible.

4.1. Machine Model

When designing an operating system the machine model must be clear, so the basis of the operating system is well understood with its functionality and its quirks. The machine for a conventional or first level compiled operating system is the hardware, i.e. the processor, which executes the code of the operating system natively.

The term Virtual Machine (VM) is common when talking about bytecode interpreters, as they fetch, decode and execute code, which is their native language. The operations however can be high level, as operating system support is normally available, at least in the conventional way of using an interpreter on top of an operating system.

In the case of PylotOS , the VM of the interpreter is the machine the operating system runs on. The PylotOS kernel is written in its native language (Python Bytecode) and the mapping between the real hardware and the virtual Python machine is handled completely in the interpreter software. The real hardware is used in conjunction with the interpreter software to emulate a virtual Python machine.

The Virtual Machine (VM) has to be able to read and alter the state of the real hardware, when needed for its operation. Otherwise the two should be two separate entities. For example the virtual machine (interpreter) may choose the activate interrupt lines or deactivate them when needed to implement some functionality in the virtual machine.

The real hardware is used for the virtual machine by executing the interpreter. It acts as mediator between the real and virtual hardware. While the real hardware uses an address based memory, the virtual machine only knows an garbage collected object space. Switching of object spaces has to be implemented in the interpreter and an interface to the Python virtual machine has to exist.

In the rest of this section CPython is described to understand the machine model.

4.1.1. CPython

The reference implementation of Python interpreters and VMs is described here: CPython. This section is based around the series of blog posts by Aknin [1].

The CPython interpreter however does two steps, when executing Python code. It will first compiles the Python code to intermediate code, so called bytecode, which can be interpreted by the Python Virtual Machine (VM). The VM behaves the same way a physical machine would - it gets its software and data from memory, and executes instructions one after the other. The VM has an internal state.

The compilation step is skipped in these considerations. The main focus is on the virtual machine and how it evaluates the bytecode instructions it receives.

Objects and Attributes

'Mostly everything in Python is an object, from integers to dictionaries, from user defined classes to built-in ones, from stack frames to code objects' [1]. Therefore objects are a central concept in the Python programming language. An object is a structure and a set of methods to manipulate the structure. The most basic object is of the type `object`, whose definition is shown in Listing 4.1. Its elements are a reference counter and a type-pointer.

Listing 4.1: C-definition of the PyObject structure

```
1 typedef struct _object {
2     Py_ssize_t ob_refcnt;
3     struct _typeobject *ob_type;
4 } PyObject;
```

The reference counter is used for garbage collection purposes, it is increased, when a name is bound to an instance of this type, dereferenced, when a name-binding is destroyed and the object can be collected, when the reference counter hits zero, as there are no names, which reference this piece of memory. The garbage collection facilities in CPython will not be described in more detail.

The second field denotes the type, of the object. This field is used by the interpreter to recognise the object. A Python object is a `PyObject*` in C, which is in turn a `void*` to memory. An object always has a type (or sometimes called class). A type in Python has a type. That means, that the `object`-type as seen in Listing 4.1 has a type, probably of type `type`. The type field in a way describes the inheritance of types.

The type of an object determines, which operations are defined on the object. The `type`-type has fields, which point to functions for certain operations on instances. These so called slots can be set explicitly, when creating a new type (via the `class`-statement), for example by defining certain methods.

A so called data descriptor takes precedence over an attribute of the object, which is stored in its private
`__dict__` dictionary. In normal circumstances the Python interpreter will do the following, when the program wants to get an object-attribute:

1. dereference the type of the accessed instance. Search the type's dictionary and its bases' dictionaries

- if a data-descriptor is found, call its `tp_desr_get` function and return its result
- if something else is found, set it aside

2. search the instance's dictionary for the name: if something was found: return it.
3. if something else was found in 1.:
   - if it is a non-data descriptor: call its `tp_desr_get` and return the result
   - if something else: return it
4. nothing was found: raise `AttributeError`-exception

The instance's dictionary is placed after the C-structure in memory. The type contains an integer, which stores the offset of the dictionary after the start of the C-structure. If the object has no private dictionary, the offset is 0.

The dictionary can also be read from Python be accessing it with the . (dot)-operator. But the name `__dict__` cannot be looked up in the dictionary. So returning the dictionary is done by a descriptor, which takes precedence over the lookup of the name in the dictionary of the instance itself.

Interpreter State and Thread States

CPython distinguishes between the interpreter state and the thread state. Every thread of executing Python code, has its own thread state and is bound to an interpreter state structure. CPython has the facilities to initialise several interpreter state structures and run as several interpreters, which share little state with each other.

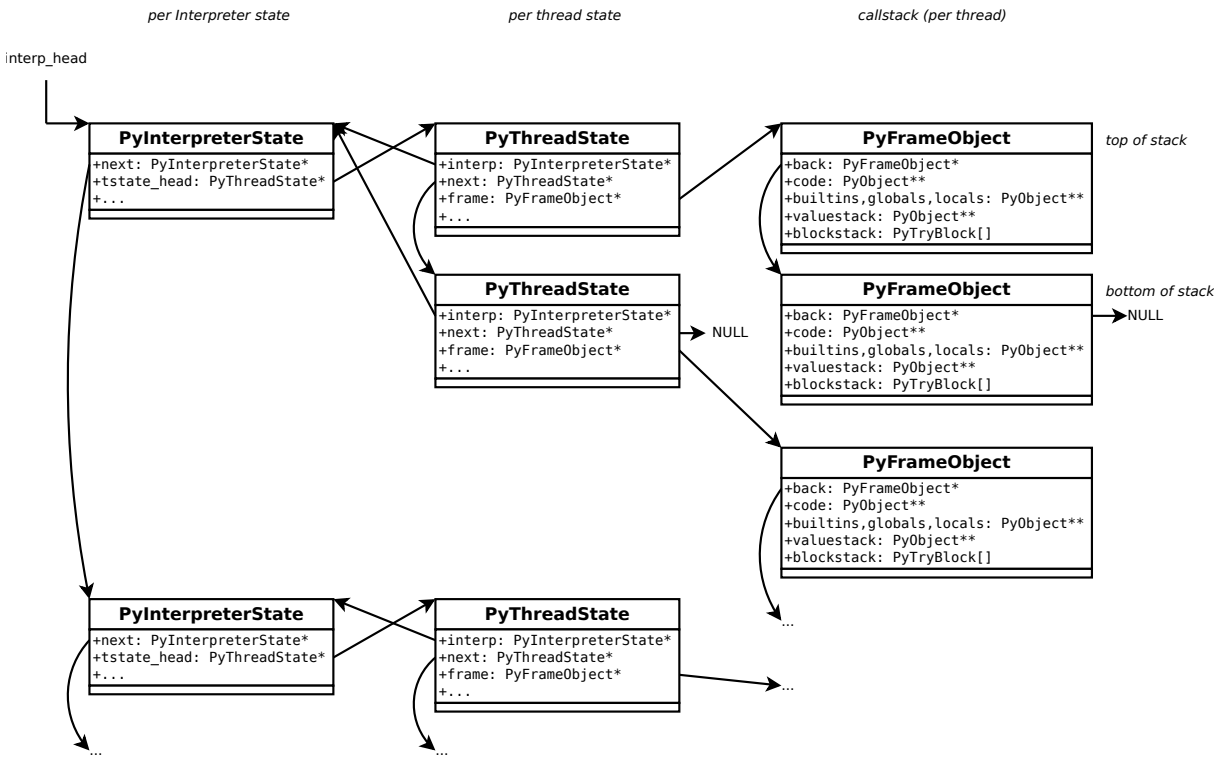

Figure 4.1.: Interpreter state, thread state and frame objects (callstack) in CPython [1]

As shown in Figure 4.1, CPython has a pointer to the first element of a list of interpreter state structures. Every interpreter state structure maintains a list of thread state structures, i.e. every interpreter has at least one thread. A thread state structure points to frame objects for the internal Call Stack. Code objects belong to the frame objects.

With threads it is possible to run Python code concurrently, but a major drawback of CPython is the Global Interpreter Lock (GIL). This lock prevents Python-code from accessing structures concurrently, whose concurrent modification may lead to corrupt or inconsistent data.

Namespaces, Scopes and Code Objects

When evaluating code, an object may be bound to a name, via the assignment (=) operator. The name is associated to a reference to the object. The reference can be altered, or the object can be altered, making it visible through all references. When binding a name to an object, the reference count to that object is increased, when deleting the reference, it is decreased. Bindings are also created through `import`, `def` or `class`.

A name is visible in its scope or block. A block is a module, function body or class definition. A namespace is an environment, where the names are mapped to the objects which they reference. Scoping dictates in which namespace a reference is sought after. When a namespace is no longer used (e.g. when a function finishes), the names are unbound.

When Python code is evaluated, there are three namespaces for the resolving of names: *local*, *global* and *builtins*. A normal load or store opcode will first look in the *local*-dictionary, then the *global*-dictionary, then the *builtins*-dictionary and raise a `NameError`-exception when the name was not found. Which opcodes are emitted, i.e. which namespaces are examined can be altered by the `global` or `nonlocal` statements.

Aknin [1] lists some special cases like nested functions and four load-store-opcode pairs, which are skipped in this discussion. The `LOAD`-opcodes push the reference of the name onto the Value Stack, the `store`-opcodes take the topmost element of the Value Stack and bind a name to that object.

Whenever a block of Python code is compiled, a so called code object is created. This code object contains not only the bytecode itself, but also additional fields, which are needed to execute the code. Nothing in the code-object may be mutable, so the code and the additional fields cannot be altered. A Python developer may choose to use literals, like `[1,2,3]`, which are used as mutable objects. These values are not stored as constants, but rather appropriate bytecode is emitted to create the object at run-time.

These additional fields contain information about the name and filename of the code object, but also about cells, local names, the number of local names, the stack size, the constants, and the zombie frame object, attached to the code object. A cell is a place for a function to store values, which are accessible for nested functions.

This interesting case, is the one shown in Listing 4.2. The function `inner` can use the value `a`, which is bound in the function `outer` although the latter is finished, when `inner` is evaluated. The bytecode compiler emits special load and store opcodes, which create the value in a special field of the code object of `outer`, in a so called cell. This cell can then be used to access the objects stored within from nested functions. This only works with nested functions, not with independent functions.

Listing 4.2: Cells and namespaces in nested functions

```
# definition of the nested functions
def outer():
    a = 1
    def inner():
        # a is visible from outer's locals
        return a
    b = 2
    return inner

# usage
print ( outer()() )

# dis.dis(outer)
  2           0 LOAD_CONST               1 (1)
              2 STORE_DEREF              0 (a)

  3           4 LOAD_CLOSURE             0 (a)
              6 BUILD_TUPLE              1
              8 LOAD_CONST               2 (<code object inner at 0x7f0e89b2e930, file "<↩
                  stdin>", line 3>)
             10 LOAD_CONST               3 ('outer.<locals>.inner')
             12 MAKE_FUNCTION            8
             14 STORE_FAST               0 (inner)

  6          16 LOAD_CONST               4 (2)
             18 STORE_FAST               1 (b)

  7          20 LOAD_FAST                0 (inner)
             22 RETURN_VALUE

# dis.dis(outer())
  5           0 LOAD_DEREF               0 (a)
              2 RETURN_VALUE
```

The Internal Stacks

For storing objects of a program, CPython uses a garbage collected heap. Inside the heap, it stores three stacks to keep certain state-data: the Call Stack, the Value Stack and the Block Stack.

The value stack is used to store objects, which will be manipulated, when certain opcodes are evaluated. For example, when a `BINARY_ADD` is executed, the topmost elements of the value stack are popped, added together and the result is pushed back to the value stack.

The call stack keeps information about the currently evaluated code objects. It is comprised of so called frame objects, which contain references to the value and block stack. A code object, which is executed, is linked to the frame object (`f_code`) it created, so when evaluating the same code object later again, the same frame object can be reused.

The frame object is implemented as a C-struct `PyFrameObject`, in *Include/frameobject.h* of the CPython interpreter. Listing 4.3 shows its definition [1].

Listing 4.3: C-definition of the frame object in CPython

```
typedef struct _frame {
   PyObject_VAR_HEAD
   struct _frame *f_back;    /* previous frame, or NULL */
   PyCodeObject *f_code;     /* code segment */
   PyObject *f_builtins;     /* builtin symbol table */
   PyObject *f_globals;      /* global symbol table */
   PyObject *f_locals;       /* local symbol table */
   PyObject **f_valuestack; /* points after the last local */
```

```
    PyObject **f_stacktop;    /* current top of valuestack */
    PyObject *f_trace;        /* trace function */

    /* used for swapping generator exceptions */
    PyObject *f_exc_type, *f_exc_value, *f_exc_traceback;

    PyThreadState *f_tstate; /* call stack's thread state */
    int f_lasti;             /* last instruction if called */
    int f_lineno;            /* current line # (if tracing) */
    int f_iblock;            /* index in f_blockstack */

    /* for try and loop blocks */
    PyTryBlock f_blockstack[CO_MAXBLOCKS];

    /* dynamically: locals, free vars, cells and valuestack */
    PyObject *f_localsplus[1]; /* dynamic portion */
} PyFrameObject;
```

Frame creation occurs, when the VM starts evaluating a code object, so, when a function is called, a module is imported, a class defined, `exec` or `eval` is used and in the interactive mode.

When an exception is raised or a `break` is called inside a loop, the exception handler or next byte code, which needs to be executed after the loop, needs to be found. The block stack exists for saving these information. It consists of `PyTryBlock`-structures, which keep information about the current compound statement state. These structures contain a type-field, which identify the compound statement, so they are valid not only for try-except-statements. The block stack is of fixed size in the frame object of the call stack.

The `f_iblock` attribute of the frame object denotes the offset of the last allocated element of the block stack. When an element is popped, this offset is decreased. A special case is a raised exception, which exhausts the block stack, without being caught. Then the previous frame objects, need to be evaluated as well.

Aknin [1] gives a code-example. The Python code is shown in Listing 4.4 as well as its bytecode representation.

Listing 4.4: A simple for-loop example for inspection of the interpreter stacks in python

```
def f():
    for c in 'string':
        my_global_list.append(c)

# dis.dis(f)
 2           0 SETUP_LOOP              27 (to 30)
             3 LOAD_CONST               1 ('string')
             6 GET_ITER
        >>   7 FOR_ITER                19 (to 29)
            10 STORE_FAST               0 (c)

 3          13 LOAD_GLOBAL              0 (my_global_list)
            16 LOAD_ATTR                1 (append)
            19 LOAD_FAST                0 (c)
            22 CALL_FUNCTION            1
            25 POP_TOP
            26 JUMP_ABSOLUTE            7
        >>  29 POP_BLOCK
        >>  30 LOAD_CONST               0 (None)
            33 RETURN_VALUE
```

Notable are the opcodes `SETUP_LOOP` and `POP_BLOCK`. `SETUP_BLOCK` is implemented to push a new element onto the block stack, whereas `POP_BLOCK` pops the topmost element from the block stack. When entering the loop, the code therefore pushes and element, when leaving it pops the

element from the stack.

Opcodes

Appendix C shows the known bytecode opcodes of CPython [6]. The opcodes can be placed into one of these categories:

- `NOP`
- Value stack modification (`POP`, `DUP`, `ROT`)
- Block stack modifications (`SETUP_FINALLY`, `SETUP_EXCEPT`,...)
- Unary operations (`POSITIVE`, `NEGATIVE`, `NOT`, ...)
- (Inplace) maths operations with two operands (`ADD`, `MULTIPLY`, ...)
- Operations on iterators and `for` (`FOR_ITER`,...)
- Operations for asynch (`GET_AITER`, ...)
- `PRINT_EXPR`
- Function calls, method calls, returning, yield (`RETURN_VALUE`, `CALL_FUNCTION`, ...)
- Binary operations (`XOR`, `AND`, `OR`, ...)
- Variations on `IMPORT`
- Variations on Store and Load-operations
- Exceptions handling and raising (`RAISE_VARARGS`)
- Basic data structure (`BUILD_LIST`, `LIST_APPEND`, ...)
- Jumps and conditional jumps (`JUMP_FORWARD`, `JUMP_IF_...`, ...)

Notably absent are opcodes which would allow direct access to memory by address or access to I/O devices. CPython normally runs on an operating system and can therefore depend its libraries on certain abstractions being available. This however is not the case for the PylotOS operating system project. Also absent are instructions to modify the internal state of the interpreter like switching from kernel to user-mode.

4.1.2. Extensibility of the Interpreter

Another requirement of the interpreter is, that is should be extensible easily. The most common way of extending the interpreter is via so called modules. A module may be loaded by the Python code at run-time and can be written either in Python or in C / C++. The Micropython interpreter allows a programmer to extend the interpreter at compile-time with C-modules [17]. This allows exporting certain functionality to the Python VM.

Another approach, which is less portable and may restrict the ability to update the interpreter to the next upstream version, is to introduce new bytecode instructions. This can be done by altering not only the evaluation logic of the interpreter, i.e. extending the new necessary opcodes, but by also changing the bytecode emitter in a way, which produces the new bytecode instructions at the correct times. However the interpreter is changed with this approach, making PylotOS incompatible with the core Python language.

4.2. Architecture

In conventional operating systems terms like user land, user mode or user space refer to the CPU mode. In PylotOS the CPU never leaves the supervisor mode or the most privileged ring, and the MMU is never used or only used for 1:1 translation of addresses. The term user land, user mode and user space refer to everything, which runs on top of the PylotOS kernel, i.e. in domains. The kernel is therefore in kernel mode.

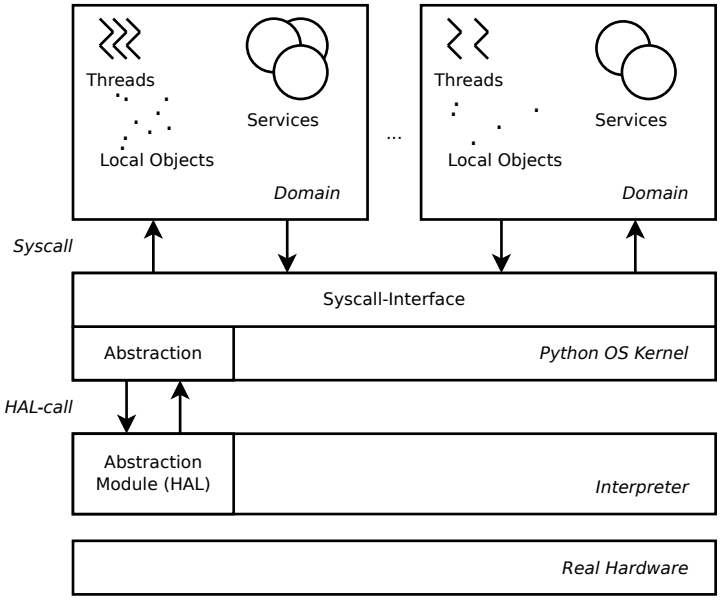

Figure 4.2.: Architectural overview of PylotOS

Figure 4.2 depicts the general architecture of the PylotOS operating system. Running on the real hardware is the interpreter, which implements the virtual machine. The PylotOS kernel is the bottommost software layer in the Python virtual machine. In general the PylotOS kernel is a microkernel. It focuses on doing as much work as necessary. The kernel itself should be as generic as possible, i.e. it should be easy to port the kernel to a different physical machine, without changing the code of the kernel.

The kernel holds the upper part of the abstraction module. The interpreter side HAL-module contains the interface for dangerous functions, which destroy the abstraction of the virtual machine, if used wrongly. The kernel has certain abstractions on top of the HAL-module, like the `Memory`-interface. It is necessary to control what user level code uses these classes. The idea of the abstraction module was to extend the interpreter with Python modules, without the need to alter the interpreter code. This was not possible and new bytecode instructions were introduced. These are outlines throughout this chapter and summarized in Section 4.7.

A main concept of PylotOS is the kernel object. A kernel object (see Section 3.8) is an object, which lives in the kernel space. The kernel can modify the object. User level code can request the kernel to interact with the object. Every object, which is shared between domains needs to be a kernel object. A kernel object is represented to the user level by a handle. User level wrapper libraries create an abstraction from these handles and wrap an object oriented view around the syscall-interface.

A domain is a process and has its own object space. The domain stores its own meta data in that ob-

ject space, so it contains not only local objects, but also thread and domain control blocks, as well as frame objects (the Call Stacks) and the Value Stacks. No user objects can be shared between domains. The domain is the unit of protection, and threads inside of domains are the unit of scheduling with some constraints. The thread has its own thread state in the interpreter, so concurrent execution of several threads at the same time by several processors is possible. An object space may be garbage collected at any time, except otherwise noted.

A service object is a kernel object which allows remote procedure calls between domains. A domain may gather a handle to a service and use an object oriented abstraction (portal) to call methods of the remote service. Domains, threads and services are explained in more detail in Section 4.4.

Communication between domains and the kernel (syscall) are implemented with the `TRAP` instruction. It traps the interpreter and instructs it to execute a kernel trap handler, which fulfils the request or denies it. The syscall interface is the only way in PylotOS to leave the domains context from inside the context and request any outside action to be taken by the kernel. This not only includes certain actions on kernel objects, this calling or different domains, but also the IPC-facilities of PylotOS .

A domain has in a certain way its own virtual interpreter, it can assume, that the interpreter does not share any state with any other domains interpreter, neither when running sequentially on a single processor or running concurrently on a multiprocessor system. The virtual interpreter stores all necessary information for executing code in its private interpreter state structure, which cannot be shared between virtual interpreters.

Device drivers are written in Python, as well as interrupt handlers. Drivers may access memory mapped device registers through the abstraction module (and in turn the kernel). They run in domains.

4.3. Kernel Objects and System Calls

The central concept of PylotOS is the kernel object. It is represented as an integer to the user domains as a so called handle. The handle is abstracted by a user level wrapper library to allow user domains to interact with kernel objects in an object oriented manner.

An object is a kernel object if and only if it needs to share state between domains or checks are needed for the proper usage of the object. For example the `Memory` objects are kernel objects, as they could be used to circumvent the security mechanisms of PylotOS . Checks for proper usage, e.g. correct addressing, region checks, ensuring access by only one thread at a time, etc. can be implemented in the kernel.

Several other objects have to be kernel objects to function properly. Service objects are kernel objects, as they are used to share state between domains - the message queue of the service. A domain may create a service kernel object, given an interface description. The kernel then allows other domains to invoke the methods of the service object through a syscall interface. The syscall interface is the abstracted to an object oriented view by the user level wrapper library. The kernel manages message queues for service objects and unblocking service threads in the target domain when necessary.

A handle is associated with a domain and certain access rights. It is possible to duplicate a handle with lower rights. It is also possible to transmit a handle over the service invocation mechanism, which will remove the handle from the sending domain and add it to the receiving one. This also holds true for the return-statement of a service call.

Domains use system calls to request any action from the kernel, be it an action on kernel objects or a generic action with no kernel object. Syscalls are implemented using the new `TRAP`-instruction. It takes two parameters, a reference to a number and a tuple of parameters. The interpreter is interrupted by the `TRAP`-instruction and a kernel routine executed, which takes care of the syscall. The interpreter state is switched to the kernel state. The kernel is able to access data of any domain, especially stacks, etc. of the calling domain. The calling thread is blocked until the syscall returns.

The kernel knows based on the number, what syscall needs to be executed and how many parameters are expected. It takes them from the tuple. The parameters denote the kernel object (if any) and additional information, modifiers and data. When the kernel accesses these data structures, it does not need to copy them into the kernel object space, as it can access any object, without violating any separation constraints.

The request may be answered immediately or deferred, for example because the target domain of a service call has other requests to process. In the latter case the kernel schedules another thread. In theory another thread of the same domain may be scheduled, however synchronisation between threads would be necessary, so no thread modifies data, which is currently used by the kernel. Also when the garbage collector runs, necessary data may be moved, so references in the kernel become corrupted. This may be solved by copying data into the kernel (at the latest before the garbage collector runs) or preventing the garbage collector from running until the request is finished.

When the syscall is answered, the kernel copies an object (graph) to the caller's object space and places a reference to it onto its value stack, so the instruction behaves like a `CALL_FUNCTION`-instruction.

Interrupting the interpreter from inside and the `TRAP`-instruction aren't standard Python. They are included in the abstraction module and will be described in further detail in Section 4.7.

4.4. Domains and IPC

In PylotOS the domain concept is the equivalent to processes in conventional operating systems. Domains cannot share any state or information with other domains. A domain has an object space, which serves as place for all local objects, frame objects, domain control blocks, etc.

A domain sees a virtual interpreter, which is not shared between domains. A virtual interpreter therefore stores its information in the interpreter state structure per domain. Each thread has its own thread state structure, which is referenced by the interpreter state structure of the domain. The virtual interpreter switches between threads by changing the reference to the currently running thread in the interpreter state structure.

A domain has its own object space, separated in memory from all other object spaces (except the kernel's one). In its memory area it contains a domain control block, which holds all information about the domain, like IDs, executing user, permissions, handles to kernel objects etc. alongside the interpreter state, as well as a list of thread control blocks. Domain control blocks are placed at the same relative address inside the domain's object space. They cannot be moved by the garbage collector. Thread control blocks on the other hand are created dynamically at arbitrary places in the object space and can be garbage collected and moved.

The isolation of object spaces is not enforced in hardware, like in conventional operating systems, but enforced in software. A standard interpreter only emits code, which references objects in its own

object space. Shared references to other programs don't exist in normal Python. However, when several domains in PylotOS are concerned - they could in theory share references. The interpreter could be extended in a way, so it actively checks every reference when it's used, whether or not the running domain may indeed access it. This is however not proposed here, as remote references between domains don't exist in PylotOS, aside from kernel objects.

Other approaches for isolating domains are discussed in Section 4.8.

4.4.1. Context Switching

For every core in a multiprocessor system one virtual processor is instantiated on boot-up. Every one of these holds a pointer to the currently used interpreter state structure. Every interpreter state structure holds a pointer to the currently used thread state structure. A virtual interpreter can switch from thread to another thread of the same domain, by altering the thread-structure pointer in the interpreter structure. It can switch from one domain to another one by modifying its interpreter state pointer.

A context switch may only occur at set times, after executing a bytecode instruction but before fetching the next one, so no context is lost. The interpreter does not need to save the program counter of the CPU as it may continue its execution at the same place in the code, however its next instruction is one of another Python program or thread.

When having several processors executing code concurrently at the same time, the threads have to be synchronised. When accessing objects, which are shared by the threads, they need to care themselves for the mutual exclusion. However, when a thread alters the object space, i.e. allocates a new object, while another one is doing the same, the object space may be corrupted. Threads need to synchronise their allocation-operations on the object space of one domain. An extension of the interpreter is necessary. More importantly when the garbage collector runs, no other thread may be executed in the same domain. Also there are times, where the garbage collection needs to be prevented.

Context switching can be done as reaction to the synchronous timer interrupt (see Section 4.1) or from cooperative relinquishing from the processor. Either way, the context switch may only occur after an instruction has been executed completely. This prohibits the thread from leaving the object space corrupt, as that would be fatal, when another thread of the same domain is scheduled.

4.4.2. Domain Life Cycle

A domain is created by the kernel. The syscall provides not only the main-code object, but also arguments to the module. Upon creation of a new domain, the kernel needs to either find an empty domain, or create a new one. Then the code object is copied to the new domain. The domain control block and the first thread control block is created. The new thread is then put into the runnable state to be scheduled in the future.

The interpreter state inside the domain control block either already exists or needs to be created. Nonetheless after creation the interpreter state should be in a state where the object heap is set up properly. The thread state should point to the first instruction of the state code object, with correct dictionaries and stacks. The lowest entry in the call stack could be an operating system exit-function, to exit the thread properly, when the main-code returns.

A domain is destructed, when a privileged user domain instructs the kernel to or the last thread of a domain is deleted. Depending on the configuration, the existing domain is emptied, i.e. the object

space emptied (except the Domain Control Block (DCB)) and the interpreter state reset to a safe level. Or the domain is deleted all together. Then the kernel would need to create a new one, when a domain needs to be created.

4.4.3. Interprocess Communication

The interprocess communication mechanism of PylotOS bases on the service kernel object, its user level representation and the client-side portal representation. The user can use them as normal objects, which behave like any other object (except the ability to change attributes). When invoking a method of a portal object it uses the syscall interface to transmit the request to the host domain of the service, where a method is executed and the return value is transmitted back to the calling domain. The user mode library acts as a wrapper between the service or portal user space objects and the syscall interface.

The User Code can use the interface of the user level library to construct the service object. It creates a class inherited from `Service` and defines methods on it. When this new class is used to create a new object from it, a service kernel object is created and the handle stored inside the service object. The domain control block contains a list of all service objects in the domain. The new service object is added into that list on construction.

Then a thread waits for requests on the `wait`-method of the service. The thread is blocked until there is a request, in which case it is unblocked and starts executing the corresponding method. The parameters for the call are copied to the object space of the host domain.

The User Mode Library provides an interface to the syscall interface of the kernel. It provides the base `Service` class and implements the constructor and the `wait`-method. The constructor uses a syscall to create a service kernel object. It lists the methods of the service object and gives the kernel this list, so it can intercept a service call before the waiting thread is unblocked. The kernel creates the service kernel object and adds it to the domain control block's list. The user mode library stores the handle of the kernel object inside the object.

When a thread calls the `wait` routine it is blocked by the kernel, waiting for its service-wait syscall to be answered. It is answered by the kernel with a reference to the called method and its parameters, when a client calls the service. The library calls the method with the given parameters. After the method is finished the service-return syscall is used to return a value to the client-domain. If an unhandled exception occurred, the service-except syscall is used to transmit an exception. Afterwards the thread waits for the next request.

The client can call methods of a service object by using a so called portal. A portal holds a handle to the service kernel object. The kernel denotes, that the handle references a portal in the access rights-flags of the handle-table of Domain Control Block (DCB) on creation in the domain. If the handle can only be used as a portal, waiting for requests, returning or deleting is not possible.

The user mode wrapper library provides an abstraction for the handle – the portal. When a handle is transmitted a portal can be created from the list of methods and their parameters. These can be retrieved from the kernel. The user code then has an abstraction of the handle, which behaves like a normal object, except it calls remote methods of the service in the host domain.

Listing 4.5 shows a sample service and its generated portal. Appendix D contains a prototypic implementation of the user level abstractions.

Listing 4.5: Implementation of a portal-call (portals are generated at run-time)

```
class mySvc ( service ):
    def __init__ ( self, x ):
        self.x = x
    def aMethod ( self, a, b, c, d ):
        raise Exception();

def pyos_serviceCall ( senderID, receiverID, serviceID, methodname, args ):
    trap ( OSServiceCall,  ( senderID, receiverID, serviceID, methodname, args ))

class _virtual_portal ():
    receiverID = x
    serviceID = y
    def aMethod ( self, a, b, c, d ):
        ret = pyos_service_call ( pyos_returnCurrentDomainID(), self.receiverID, self.↩
            serviceID, "aMethod", (a,b,c,d) )
        if ( ret[1] != None ):
            raise ret[1]
        else:
            return ret[0]
```

When a domain invokes a method of a portal, a new request is placed inside the kernels message queue for the service object. If it is the first one, the service thread's syscall is answered, by copying the parameters into the object space of the hosting domain and returning references to the requested method and the parameters. Otherwise this is done, when it is the first request in the message queue.

The parameters are copied, only when the request is the first one. This might be risky, as other threads of the client domain may alter the data, so synchronisation is necessary. When the garbage collector needs to run, there are two possibilities: either the parameters need to be copied into the kernel or pre-maturely into the target domain, or all running threads of the domain are stopped until the service call is answered.

Destruction of a Service is initiated by a syscall of the hosting domain. Then the message queue is emptied and the syscalls of the callers are answered with an exception. Every new request is answered with an exception. From the moment the service was destructed onwards, no more requests may be placed in the message queue. The currently executed request however is finished properly. When the service thread calls the service-wait syscall again it is instructed, that the service no longer exists. The service thread may then exit or do other work.

Parameters are necessary to modify the method invocation or to present data to the service. However it is not as trivial to just copy the referenced object when handing over parameters. The service may then choose to use any reference on that object, including functions. Also the object is of no use, without the type-definition of the object. A major problem is here, that the naming-dictionaries of the client-domain must be left in the client domain and must not be transferred to the host domain.

So the routine copying the parameters to the target domain, needs to act recursively and copy not only the referenced objects, but also the objects referenced by that, including their type, if these are not already present in the target domain. All references need to be updated to deal with the changed memory locations.

When copying handles to kernel objects two policies are possible. Either duplicate them and transmit them, so the service object may use the kernel object. Otherwise the handle may be copied, without adding it to the table of kernel objects of the target domain, essentially making them unusable to the target domain.

4.4.4. Name Service

Additionally to the bare functionality of the interprocess communication in PylotOS a so called name service can be implemented as a Domain. This domain creates a service object and registers it at start-up with either the operating system or the init-Domain. A handle to it can then be passed to every domain on creation, so it may access other domain's services.

The name service only needs three methods: a register-method, an empty-method and a lookup-method. The name service may be implemented as listed in Listing 4.6. It keeps a list of services together with their registered names. A name could in theory hold more than one service object.

Listing 4.6: Prototypic implementation of a Name Service

```
1  from service import *
2
3  class NSNameTakenError ( Exception ):
4      pass
5  class NSNameNotFoundError ( Exception ):
6      pass
7
8  class NameServiceEntry ( object ):
9      def __init__ ( self, prtl, name ):
10         self.prtl = prtl
11         self.name = name
12
13 class NameService ( Service ):
14     def __init__ ( self ):
15         self.prtls = []
16
17     # register a portal with a name
18     def register ( self, prtl, name ):
19         for s in self.prtls:
20             if ( s.name == name ):
21                 raise NSNameTakenError
22
23         prtlEntry = NameServiceEntry ( prtl, name )
24         self.prtls.append ( prtlEntry )
25
26     # look up a portal by name
27     def lookup ( self, name ):
28         for s in self.prtls:
29             if ( s.name == name ):
30                 return s.prtl
31
32         raise NSNameNotFoundError
33
34     # clear the associations with a name
35     def clear ( self, name ):
36         for s in self.prtls:
37             if ( s.name == name ):
38                 self.prtls.remove ( s )
39                 return
```

A lookup of services is similar to opening device files in Linux. The process knows the name of a service, like *dns* and looks it up in good hope, that it is the correct service. Additional steps can be undertaken to ensure, that the correct service was found, like checking the portals methods, but that should not be necessary. A list may also be created for standard names corresponding to standard functionality, like the name 'ip4' may reveal a service, which is the IPv4-layer in the network stack. For additional grouping names could contain slash-characters (/) for separating semantical units of the name.

When the name service transfers a handle to a service object to a client, it first duplicates its own handle, so it still keeps a handle. It then returns the newly created handle to the client. When the service was destructed by the host domain, the duplication fails.

File Systems A virtual file system can be implemented as a service for other domains to use. There should be a single method `open`, which allows a domain to open a file. This method locates the file and brings the device in a state, which allows for access to that file and creates a service object for the file. A handle is then returned to the calling domain, and may be used to read, write, ioctl, seek, tell, etc. on the file. Close decrements the reference counter of portals to the service object. If it hits 1 (the one held by the virtual file system), the service can be destructed.

The virtual file system needs to invoke the functionality of services of device drivers and drivers for file systems, maybe also drivers for file or disk encryption. The basics of device drivers are discussed in Section 4.5. The details of file systems and the driver stack for accessing files are not explored in this thesis.

4.5. Device Drivers

The idea of a driver is to have a modular approach to device support, so that a single operating system kernel can load different drivers depending on the configuration of the hardware it runs on, and may control several different types of devices. In 4.3BSD the configuration step was at compile-time, in more modern operating systems, the OS discovers devices on boot-up and loads drivers at runtime.

With this in mind, the driver infrastructure in PylotOS is completely held in user space. Similarly to Fuchsia, a driver is loaded after making sure, that it may control a found device. It is loaded either into its own Domain or in a common driver domain.

The driver sets up a `DriverService` object, for the I/O Manager to use, when certain events happen, like a new device is detected, which the driver may control, a device is unplugged, or the driver is to be unloaded. When a device is bound to a driver, i.e. the driver receives the task of managing the device, the driver instantiated a `DeviceService`, which user applications may use to interact with the device. Certain operations may be implemented, which allow using the device intuitively. The term user application means in this case user provided programs, as well as other device drivers or the virtual file system.

Driver Service objects are not registered at the name service. A user program cannot use a driver directly, as only the I/O Manager needs to interact with drivers. Therefore every `DriverService` object is registered with the I/O Manager. When a driver is loaded, the I/O Manager's service handle is passed to the device-driver alongside the global name service handle.

The overall driver architecture is depicted in Figure 4.3. The drivers and the I/O Manager are placed inside domains. Both have service objects, the device drivers `DriverService`s and `DeviceService`s (per device). The I/O Manager holds its own service, which allows communication with the I/O Manager. The drivers need to escape the virtual machine, i.e. they need access to the hardware, which is illustrated by a dashed line and arrow to the real hardware.

4.5.1. Driver Life Cycle

When it is decided, that a new driver needs to be loaded it is necessary to find the new driver. In Fuchsia, the drivers are in ELF-format, where a header-field contains the IDs of supported devices. Similarly, PylotOS has its drivers in some format, which allows annotation or header-fields so the I/O Manager does not need to load the driver to find out that it does not support the found device.

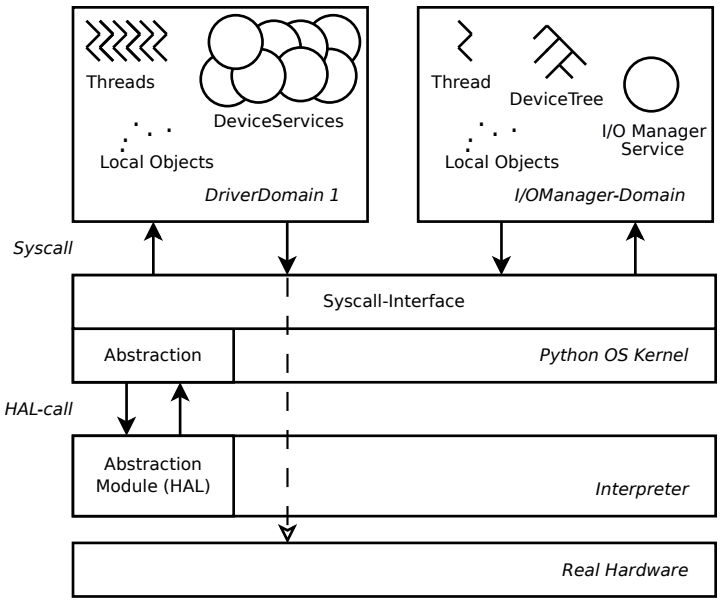

Figure 4.3.: Overall driver architecture in PylotOS

When the necessary driver was identified it is loaded into a new domain or an existing driver domain. The decision whether or not a new domain is created is done by an arbitrary policy in the I/O Manager. Separating drivers allows them to work more independently, loading them together into one domain reduces the waste of memory and domain-switches.

The driver is loaded, a new thread created for the driver and the driver code is executed. The initialisation of the driver needs to care about making it runnable and usable, as well as creating a `DriverService` object, which is then registered with the I/O Manager. If no service was created or an exception was raised, which the driver did not catch itself, the initialisation is assumed to be faulty and the driver is unloaded again. At the initialisation stage no communication or probing for devices takes place.

When the driver is initialised correctly and the I/O Manager received a handle to the driver service, it will start to bind devices to the driver, via the driver's `bind`-method. The device is represented with its bus-specific identifiers and locations, as well as features, power states, etc. - everything the bus driver may have found out about the device in the enumeration stage (see Section 4.5.4).

On binding, the driver creates a `DeviceService` object. The driver raise an exception, or return an error code if there is a problem, either with the data structure or with the device binding. On success, the device is represented by a `DeviceService` object, which is placed inside the device tree of the I/O Manager.

The `DeviceService` object is registered at the name service, so user applications are able to use the device by the use of the service's methods. Before using the portal however, the user code has to call the `open`-method to give the driver a chance to bring the device in a usable state. When the user code is finished with the device it calls `close`. The `DeviceService` object uses `open` and `close` to reference count. When all portals to the service are closed it is safe to unbind the device from the driver and delete the service.

4.5.2. Driver Interface

A `DriverService` object is only used by the I/O Manager. The driver needs to provide the following hooks:

- `bind` is used for offering the driver a device to manage. If the method succeeds, the driver has created a new `DeviceService` object, which will become a child device of the one passed as argument to the `bind`-method.
- `enumerate` is only useful on bus drivers or proxy drivers, with a reference to a device as parameter. It enumerates the devices, which are attached to the device. With the help of the I/O Manager, new drivers are loaded and the devices are taken into operation.
- `release` is invoked, when the driver is no longer needed before unloading the driver. This occurs, when the last bound device of that driver is destroyed.
- `unbind` is used, when a device is removed or about to be removed to instruct the driver to let go of that device and instruct all its services to stop communication with the device.
- `softexc` is used, to let the driver process an interrupt on normal priority. The interrupt handler may choose to place an interrupt request package in the kernel memory and after returning the kernel indicates, that there are unanswered softirqs in the queue to the I/O manager.
- `power` is called, when the power state of any device, which is controlled by the driver is changed. The driver is therein instructed to change the power state, i.e. powering the device down or up.

The `init`-method of Fuchsia is not necessary, as the global initialisation of the driver is done when it is loaded. `enumerate` is placed in the `DriverService` as it is exclusively used by the I/O Manager on start-up or enumeration at run-time, but may not be used by user domains.

Similar to the Fuchsia protocols several classes of devices can be defined. The driver then needs to implement certain interfaces in the `DeviceService`. Appendix E shows a prototypic implementation of a GPIO-driver for PylotOS .

4.5.3. Low Level Drivers

A driver needs low-level access to the hardware registers. In C a programmer would either create a `struct` to map the registers and set a pointer to the base address of the peripheral or use an `enum` to list all registers and access a pointer to the base address by using the values of the enumeration as index.

Both of these ways are not possible in Python. Mapping objects to a certain address is not possible in Python. Micropython can make use of assembly-instructions, which is fine for very small operations like disabling interrupts on the processor. But accessing memory mapped I/O ports would be way too cumbersome to do in inline assembler.

Therefore device drivers in PylotOS can make use of a class `IORegister`, which implements the details and allows a driver developer to use it like a C-struct. The Python programmer creates a new class, with an attribute `baseAddress`, which sets the base address for the memory mapped I/O ports. Then the class needs a set of `IORegister` objects, which all take an offset from the base address in byte. Accessing the `IORegister`s can then be done with the normal = operator.

When writing something to an attribute, which is an `IORegister`, it will call the `__get__` or `__set__`-method respectively. These will read the `baseAddress`-attribute of the instance class including the `IORegister` into it and add the offset onto that. The user land wrapper around `Memory` kernel objects is then used to access certain addresses in the system.

Listing 4.7 depicts a bare-bones example of such a structure.

Listing 4.7: Bare-bones example for a low-level driver structure

```
1  from userland_memory import *
2
3  # IORegister describes an offset to the base address set by the using object
4  class IORegister ( object ):
5      def __init__ ( self, offset ):
6          self.offset = offset;
7
8      # get the value from the base address + offset
9      def __get__ ( self, instance, owner ):
10         if ( instance == None ):
11             raise Exception("No Instance?");
12         return instance.memory.getUint32 ( self.offset );
13
14     # set a value to the base address + offset
15     def __set__ ( self, instance, value ):
16         if ( instance == None ):
17             raise Exception("No Instance?");
18         return instance.memory.setUint32 ( self.offset, value );
19
20 # a simple device class
21 class MyDevice(object):
22     # control register located at 0x20000000 (if base=0x20000000)
23     control = IORegister ( 0 );
24     # value register located at 0x20000004
25     value = IORegister ( 4 );
26
27     # set the device in a controlled state on initialisation
28     def __init__ ( self, base ):  # assume base=0x20000000
29       self.baseAddr = base
30       # memory block from 0x20000000 to 0x20000008
31     self.memory = Memory ( self.baseAddr, 8 )
32         self.control = 0x12;
33         self.value = 0x00ff00ee;
34
35 dev = MyDevice ();
36 print ( dev.control );
37 dev.value = 0x00ff00ff;
```

4.5.4. The I/O Manager

The I/O Manager is not only responsible for enumeration of devices, but also plug and play and power management. When it is started, it begins with enumeration of devices, i.e. it loads the bus driver of the system bus and instructs it to enumerate all attached devices. All the found devices are represented by a bus specific identifier. When a new device is found, the I/O manager looks through its known drivers and finds the driver(s), which is able to control the found device.

The found driver(s) is loaded, it creates a service object for itself. The I/O manager then offers the device to the driver (through its `bind`-method). If the driver takes the device and creates a `DeviceService`-object it is placed in the device-tree, which is held by the I/O manager and the service is also made public to the name service. The driver is then instructed to enumerate all attached devices.

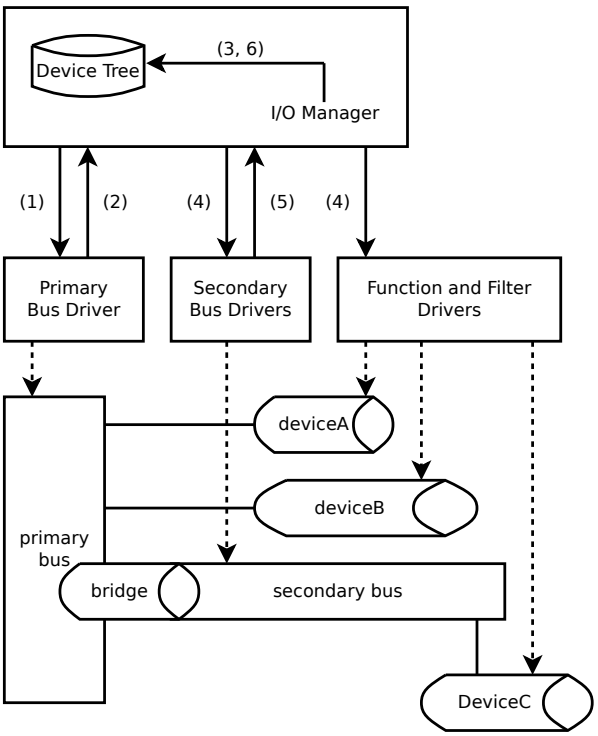

Figure 4.4.: Plug and Play architecture of PylotOS
(1) The I/O Manager loads the primary bus driver and instructs it to enumerate all attached devices.
(2) The primary bus driver reports a list of detected devices. The I/O Manager looks through its list of drivers for one supporting the recognised devices.
(3) The I/O Manager places `DeviceService` portal objects for every detected device in the device tree.
(4) Secondary bus drivers, function (and filter) drivers are loaded. Bus drivers are instructed to enumerate all devices on secondary buses.
(5) Secondary bus drivers report all attached devices; Step 4, 5 and 6 may occur recursively until no new devices are found.
(6) The I/O Manager places `DeviceService` portal objects for every detected device in the device tree.

Figure 4.4 shows the Plug and Play-mechanism graphically. The I/O manager not only cares about enumeration of devices at start-up, but also about propagating not only user requests like power-state changes along the device tree, but also for removing devices in an orderly fashion. When the user instructs the system, that she wants to detach the device, the device is de-initialised via the `unbind`-method and the `DeviceService` deleted. If it is removed surprisingly, then it is deleted without de-initialisation of the device.

The I/O Manager uses the `power` method of the `DriverService` portal to issue commands to change the power state of a device. The according arguments contain additional information about the following power state. The driver will report back if that state change is acceptable and change the state if so. It may also choose to set the power state of the device to a sleep-state when the last handle to its service object is closed. The device will be woken up, when the `open`-method of the `DeviceService` is called.

4.5.5. Interrupt Handling

A very important part of I/O is the ability to react to interrupts by the hardware. As discussed in Section 4.1 interrupts of devices interrupt also the Python virtual machine. So the interpreter catches interrupts, stores its context and begins execution of an interrupt handler, which the kernel previously made public to the interpreter. The interrupt handler starts in the Python virtual machine from a separate interrupt context with kernel context. Kernel actions may need to shield themselves from being interrupted, so the interrupt service routine could assume a consistent kernel object space.

The PylotOS kernel maintains a dictionary interrupt handler associated with their lines. This dictionary is the first step in Python to decide which interrupt handler to call. The driver's interrupt routine is called, which may silence the interrupt, service the device and return afterwards. The driver interrupt handler also runs in the context of the kernel.
The interrupt handler may find it suitable to defer some action to a later point in time, as it may involve more computations, which are however less important than other code or other I/O exceptions. The interrupt handler can enqueue a softirq (cf. softirq in Linux 2.6 in Section 3.4.1 or DPC in Windows in Section 3.5.1) by placing an interrupt request package and returning. Every driver has a thread, which waits on a single interrupt kernel object. When an interrupt request package is found, the corresponding driver is determined and unblocked. The wait-syscall is then answered with the interrupt request package.

Adding to a list contradicts the tips given in [11]. The kernel cannot be interrupted, so a consistent object space can be assumed, when the interrupt handler is active. However allocating dynamic memory may lead to the garbage collector running, which may be catastrophic for the response times of the system. To prevent that, a number of empty objects are created to be filled out in the interrupt handler, without creating new objects in the object space. Another way would be to not use a list, but a memory region or `bytearray`.

This allows the interrupt handler to do only the bare minimum of work and return, as any longer computations can be deferred. When the garbage collector collects the a driver domain, it needs to mask the interrupts for that domain and update the IDT references in the kernel, as the code objects for the interrupt functions may have been moved.

The driver's interrupt handler may access the driver's memory area and use its dictionaries to find objects. The driver has to make sure, that accesses to shared data structures are secured by disabling the corresponding interrupts lines.

4.6. Boot-up

When powering on the hardware, the interpreter is started and takes control over the hardware. It sets up the C-level interrupt service routines and the timer (if needed and if it wants to emulate a virtual timer device). It sets up its context, i.e. bringing the CPU in a usable state, initialising caches, and allowing the CPU to access unaligned data [7]. The global garbage collector for the whole memory is brought up and thread and interpreter states per core are created. For now all cores but one execute an endless loop. Then the Python virtual machine is started by executing the boot-up section of the kernel.

What the kernel does depends on the configuration. It may choose to set up several empty domains at start-up and fill them later as they are used. This would prevent the garbage collector from running when a domain needs to be created and the memory is not present for a domain-object (with its

object space). If they are created on boot-up, their memory regions can be initialised by the garbage collector and they can be placed into a list of available domains. The DCBs are also created now, alongside the interpreter states for the domains. Alternatively domains will be allocated when their creation is requested.

The kernel sets up its general interrupt handler, so when the interpreter sends an I/O interrupt it gets handled and dismissed. The kernel allocates a message queue of interrupt requests and also allocates a set amount of empty packages with empty byte-arrays. Appendix F shows a basic implementation of the generic interrupt handler. The dictionary, which contains driver interrupt handlers is empty at start-up. Logging has not been discussed in this thesis. The generic interrupt service routine is set to the interpreter, so the first level ISRs call the generic Python ISR.

Next the name service domain is started. It serves as the init-process in PylotOS as all domains need a handle to the name service. It starts the I/O Manager. The I/O Manager and the device drivers need the handle to the name service to register the portals to `DeviceService`-objects. The I/O Manager starts the enumeration-process and builds its device-tree.

After starting the driver subsystem, the system is in a working state. The name service then creates and starts the init-domain. It may then read an init-script and decide which actions to take, which code to load, etc. Threads can then be scheduled to run on the cores of the system.

4.7. Abstraction Module

As seen in Section 4.1 the interpreter is a mediator between the real hardware and the virtual Python machine. The virtual machine executes code and presents the code running on it with an interface for easier access. In theory it would be possible to export every variable in the interpreter as 'register' to the virtual machine, but that becomes tedious quite quick and leads to every patch changing these exports, whenever global state is edited.

Therefore a set of functions and classes is defined, which extend the interpreter, without interfering with the base logic of the interpreter. Some of these functions and classes are exported for use in the Python virtual machine, others only serve for accessing certain registers of the real hardware. Some changes in the logic of the virtual machine are however necessary.

The abstraction module in the interpreter basically serves as HAL, exporting very basic functionality to the Python virtual machine. These help control either the interpreter and in turn the virtual machine itself, or the real hardware, where necessary. On top of the HAL, the operating system defines a set of classes and functions to form prettier abstractions to the ugly interfaces of the virtual machine.

In the rest of this Section, the higher level abstractions are discussed and the functionality of the HAL is outlined. The OS-abstractions are not usable as is in user land, however the user mode wrapper library makes it available through the syscall interface and kernel objects.

Memory Access One of the most important functionalities is the memory abstraction. The Python virtual machine only knows the concept of an object space, i.e. it is impossible in pure Python to access a certain memory address, but it is possible to create and alter objects in that object space. The interpreter then takes care of allocating memory, accessing data and deleting the object (garbage collecting), i.e. mediating between the two machines.

Access to addresses is needed especially when accessing memory mapped device registers. The `Memory` class allows requesting a piece of memory, writing and reading data to any address within the allocated memory area. The class can be used in the kernel, and referenced via the use of a handle from the user land. The object only holds a reference to the starting point of the memory piece and its length, and does not allow array-like access to the piece of memory, but the methods allow predictable write and read operations. The downside is, that it's tedious and only allows writing and reading integer-values. A function can be implemented in Python, which uses a `Memory` object and writes blocks of data into it.

Listing 4.8 shows the principle interface of the `Memory` class. A version, which increased an offset after a write-operation can be derived from that. A release-method is not shown.

Listing 4.8: Prototype memory buffer abstraction

```
1  from hal import *
2
3  class MemoryTakenError ( Exception ):
4      pass
5  class MemoryInvalidOperandsError ( Exception ):
6      pass
7  class MemoryOutOfBoundsError ( Exception ):
8      pass
9
10 class Memory ( object ):
11     m_taken = []
12     def __init__ ( self, start, length ):
13         self.start = start;
14         self.length = length;
15
16     @staticmethod
17     def request ( start, end ):
18         # is the requested field of size 0?
19         if ( start == end ):
20             raise MemoryInvalidOperandsError()
21
22         # swap start and end if necessary
23         if (start > end ):
24             tmp = start;
25             start = end;
26             end = start;
27
28         # check for permissions
29         # ...
30
31         # check if the memory is taken already
32         for t in Memory.m_taken:
33             if ( not(t[0] > end  or t[1] < start) ):
34                 raise MemoryTakenError()
35
36         Memory.m_taken.append ((start, end))
37         return Memory ( start, end-start )
38
39     # set a 32 bit integer
40     def setInt32 ( self, offset, val ):
41         if ( offset > length or offset < 0 ):
42             raise MemoryOutOfBoundsError()
43         hal_setInt32 ( start+offset, val )
44
45     # ...
46     def __str__ ( self ):
47         return "Memory Object: %d - %d"%(self.start,self.start+self.length)
```

Low level access is handled by the HAL in C and exported to the Python virtual machine. Operations for writing specific values to specific memory addresses must exist. It must be ensured, that only the high level `memory`-module loads these low level routines. A permission check is not necessary then on the lowest level.

The `Memory` class may even be used as a means to create shared memory. This however is dangerous and not intuitive for a Python programmer, as it is impossible to allocate memory in the distributed memory by creating a new object, and concurrent access to the same memory region may lead to inconsistent data. This would also undermine the Portal/Service IPC mechanism and the protection between Domains.

Object Spaces Every Domain in PylotOS has its own object space. The kernel could include the starting and end-positions of the heaps in the interpreter state and change the values in the interpreter state when needed (e.g. when the object space is grown). Creating an object space and initialising the garbage collector is necessary too, as the garbage collector needs meta data for it to work.

An easy approach is to use the `OSpace`-class. A principle implementation is shown in Listing 4.9. Mapping the `OSpace`-object onto the `Memory`-object is in fact also possible.

Listing 4.9: Prototype heap / object space abstraction

```
1  from hal import *
2
3  # check, so only the kernel may use this module
4  # ...
5
6  # switch back to kernel OSpace (the whole object space)
7  def switchToKernelOSpace ():
8      hal_switch_to_kernel_ospace ()
9
10 class OSpace ( object ):
11     def __init__ ( object ):
12         self.buffer = hal_memory_buffer();
13
14     # switch the system to use this heap
15     def switchTo ( self ):
16         hal_switch_to_ospace ( self.buffer );
17
18     # garbage collect this heap
19     def garbageCollect ( self ):
20         hal_garbage_collect ( self.buffer );
```

The memory buffer is implemented as an array in C in the HAL. The interesting methods are the ones switching the object space. Assuming the interpreter state contains heap-pointers in the interpreter, these two addresses need to be set to the starting and end points of the underlying memory buffer, when creating a new domain. These addresses are also used by the garbage collector.

The kernel also has to be able to get references from the name-dictionaries of a domain and has to be able to access objects in the object space of domains. Also it has to be able to create objects in their object spaces and copy data into them.

Threads and Domains The OS needs to be able to switch domains, i.e. switch the interpreter state of the virtual interpreter. When switching threads, only the thread state pointer in the interpreter state needs to be altered.

Switching the whole thread state has the downside, that per thread a thread state has to exist, which increases memory consumption. The upside however is, that it is efficient to change threads, as a single pointer has to be changed in the interpreter state. The same is true for switching between domains and changing the interpreter state pointers in the virtual interpreters.

Syscalls For syscalls a `TRAP`-bytecode needs to be introduced. When executed, the interpreter will save its state (i.e. note its current state in the interpreter and thread state) and switch to a different

trap state structure. The interpreter then begins to execute a code object, which has been previously made public to the interpreter by the PylotOS kernel. The reference to it is known to the C-code of the interpreter's trap handler. The `TRAP`-bytecode might execute a software interrupt instruction, but could also execute the trap handler directly. The interpreter needs to switch to the kernel context.

The code of the kernel takes two references from the process' value stack: an integer object to decide which syscall was requested and a tuple of parameters to the syscall. The trap handler then decides which action to take, does the requested action and places the result value back on the process' value stack, before restoring its state and returning to normal operation.

Interrupts are caught the same way as the `TRAP`-bytecode, but interrupts are not synchronous. The interpreter can make them synchronous by disabling all interrupt sources while a bytecode instruction is executed and only check after one was completed. For the interpreter's interrupt handlers it still needs to store the registers of the processor to return to the interrupted piece of code.

The interpreter needs to manage an interrupt table, so an interrupt service routine can execute Python code after saving the context. These pointers to Python functions can be easily placed inside the abstraction module, alongside functions to set, get and enable / disable them. There could be three of them: one for all I/O interrupts, one for traps and one for any CPU exception.

I/O Instructions The virtual machine loses its separation between the real hardware and the virtual machine, when it comes to I/O instructions. These instructions need to be executed, on certain processors to access device registers. For the sake of separating the two concepts real hardware and virtual Python machine that is unfortunate. Implementation can be achieved by exporting functions in a generic manner, which wrap the I/O instructions. For example `hal_io_in` may execute the *in*-instruction with a register for the I/O port address and one for the value read from the port.

Virtual Timer When a pre-emptive scheduling policy is used, a virtual timer needs to be implemented, which only interrupts the virtual machine, after an instruction has been finished. This can be done based on a hardware timer, storing the information, that it interrupted and looking for the flag after every bytecode execution. The virtual timer needs to be implemented in the interpreter.

4.8. Other Approaches

In the previous sections an architecture design for PylotOS was shown and described, with a domain concept, signal-portal-IPC and a driver model. This section outlines other approaches, which have been dismissed. Nevertheless these ideas can prove valuable for further research.

4.8.1. Global Object Space

In Python there are no private members (of classes or modules), but the usable names are represented in the name-dictionaries (`builtins`, `globals`, `locals`). If a name is not in one of these dictionaries a process may not access the value behind it.

Having separated object spaces is not necessary when solely the ability to alter other process' objects is concerned. Assuming a process model, where code is loaded into one global heap, therefore all threads operate on the same heap. It is possible to isolate the threads completely (from the Python view) by limiting their view on objects on the heap, i.e. separating their name-dictionaries. This would mean they can only access their own objects or ones, which are explicitly shared with them.

Separating processes like that suffices to isolate them. Service objects can still be used, by having a name service, known globally, which allows listing (or searching for) other services. A remote reference is then the normal Python reference to the service object, allowing the calling thread to execute code of the service object directly without the hosting process knowing about that.

In theory this would suffice, however it yields major drawbacks. The threads would need intensive synchronisation, maybe even to the point that services would only be executable by one thread. Also no more than one thread may ever allocate a new object at a time. The garbage collection runs would take long times, which would also prove consequential for interrupt handlers.

Also handing out references to objects of other processes has the drawback, that the client-process may alter the object, like set other function pointers for its methods, invading the service object completely. Every other process, which would call the methods of the service object afterwards would then not call the original code, but the injected code by the evil process.

This effect can be mitigated by placing the portal objects in between the service and the client process. However hiding a service object behind a portal is not possible as all members are public in Python, so the client code can still access the reference to the service directly.

4.8.2. Syscalls via Exceptions

A different approach to the `TRAP`-bytecode is to use exceptions to return control to the kernel. Whenever a user-domain requires an action from the kernel it would raise special exceptions, which only the kernel should handle. Nothing would stop user-code from catching these exceptions.

The exception is raised, the callstack traversed for the exception handler and execution continues in the kernel. In theory, when catching a syscall-exception, the kernel stores the user-stacks and context (thread state), does the action and restores the context, by placing the popped stack frames back on the top of the callstack and returning. This would work for the callstack.

The value stack contains the currently pushed values. These values are used for virtually everything in Python, from adding two values (pops the two topmost elements, adds them together and pushes the result), to function calls (pops the topmost element and uses the `call`-method of that object).

Listing 4.10 shows simple Python code, which calls functions without any parameters and with one parameter including its disassembly.

Listing 4.10: Function call in Python bytecode

```
1  def outer():
2      a = 12
3      inner ()
4      inner ( a )
5
6  def inner ( a ):
7      return a
8
9  # dis.dis ( outer )
10   2           0 LOAD_CONST               1 (12)
11               2 STORE_FAST               0 (a)
12
13   3           4 LOAD_GLOBAL              0 (inner)
14               6 CALL_FUNCTION            0
15               8 POP_TOP
16
```

```
17   4           10 LOAD_GLOBAL              0 (inner)
18               12 LOAD_FAST                0 (a)
19               14 CALL_FUNCTION            1
20               16 POP_TOP
21               18 LOAD_CONST               0 (None)
22               20 RETURN_VALUE
23
24 # dis.dis ( inner )
25   2            0 LOAD_FAST                0 (a)
26                2 RETURN_VALUE
```

The first line loads the constant 12 and stores it into the local object a. Then the global name `inner` is resolved and the reference is put onto the Value Stack, called without any parameters and the return-value is ignored (`POP_TOP`). Lastly the same function is called with one parameter (put onto the stack: first the function, then the parameter). At the end the return-value is generated (`None`).

When taking into account, that resolving a function name needs a spot on the value stack, a major issue becomes obvious: when receiving a syscall-exception, virtually doing anything will destroy the Value Stack-values, which were put in place by a domain-code, making returning to the domain impossible. Even worse, the `RAISE_VARARGS`-instruction places the exception as well as two other values onto the value stack. This is valid in Python, as returning from an exception is normally not possible. This would trash values on the value stack even if the function-problem would be solved.

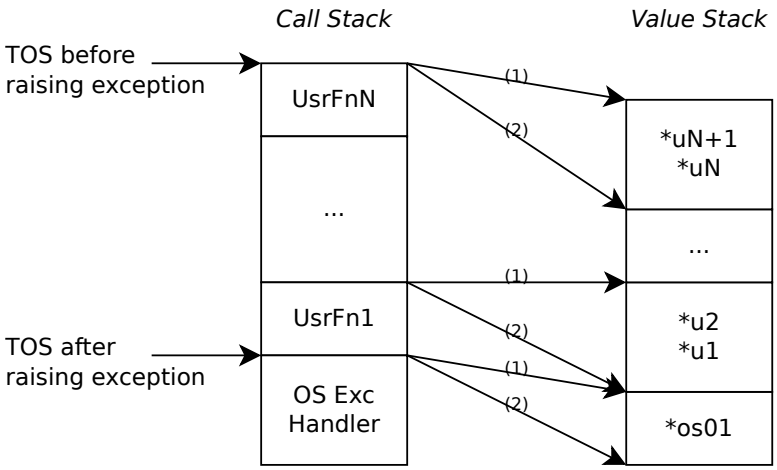

Figure 4.5.: Syscalls and the Value Stack
(1) Top of the value stack from the stack frames' point of view.
(2) Bottom of the value stack

Figure 4.5 illustrates the situation. When raising an exception the Top-of-Stack is lowered, so all frames above the exception handler become dangling, and so does the Value Stack. In normal Python operation returning to the previous state is not possible, so there is no need to care about old values on the Value Stack, three of them are already overwritten with the exception, so state of the process is lost.

A way to solve the issue with calling a function, e.g. to switch the stack, is to persuade the interpreter to emit bytecode for pushing a single None-value on the Value Stack, before putting any arguments or the reference to a function code onto it. After returning the `None`-value would need to be popped and ignored.

This would mean a single value can be pushed on the stack. If the store-domain-stack-function needs parameters even more memory needs to be wasted. This method is wasteful for the mem-

ory of the system and increases the cost of calling a function. Another approach could be to integrate logic into the interpreter to evaluate which exception was raised and allow the interpreter to switch stacks, when syscall-exceptions where raised. This would be similar to the behaviour of some CPUs, but would need extensive modification of the interpreter.

A way to resolve the issue with handing the exception over the value stack, is to store the exception alongside the other two values in the thread-state instead of the value stack. This would however limit the expressiveness of the exception concept and involve major interpreter patches.

This approach has been discarded, as the problems are hard to overcome cleanly and without altering much of the interpreter. This way of implementing syscalls was seen as ham-fisted, so a dedicated `TRAP`-bytecode was introduced and an interrupt table created at the interpreter level.

`YIELD_VALUE`-instruction

Python allows the creation of co-routines, which allow for returning from a function while storing the context. The function can then be used like an iterator, where the next value is calculated on the call to `next`. It would be possible use that as syscall, if and only if every syscall was requested from the bottom-most function over the OS layer. Constraining the logic of user programs is possible but unwanted, so this approach was discarded.

I/O Exception Handling

Another way of handling interrupts was discarded as infeasible. The idea was to represent interrupts as exceptions inside the Python virtual machine. It would work similarly to the syscall mechanism proposed, but still the problem of returning was immense as parts of the value stack where lost and even worse an I/O exception could come at any point in time in the execution of a bytecode instruction, meaning more context would be lost or it would need to be stored.

4.9. Closing Remarks

This chapter described the overall architecture of the PylotOS operating system. It has outlined kernel objects, domains and the service-facilities. On top of that, the driver infrastructure was outlined. The syscall interface into the kernel has been explained and the machine model and the abstraction module were discussed.

This final section goes through example procedures, normally found in operating systems, and explains several of the key features of PylotOS . The list of examples is of course not extensive and does not cover every aspect of the system, but tries to make the system architecture more clear.

4.9.1. Blinking LED

A simple example is using a user domain and a single driver to let a LED blink on one of the GPIO-pins, in this case of the Raspberry Pi Zero. Section 4.6 shows the hardware set-up. The LED is connected to a GPIO-pin and the ground-connection with a resistor in series.

Assuming the LED attached to the pin number 7, the domain calls the name service to return the handle to the GPIO-device service. Remember, calling the name service means in fact using the user mode wrapper library to do a `TRAP` into the kernel and request a message to be enqueued at the name service's message queue. The name service thread will be awoken, if not already in runnable-state. It has waited at a wait-for-service-calls syscall. The message of the caller will be transferred to

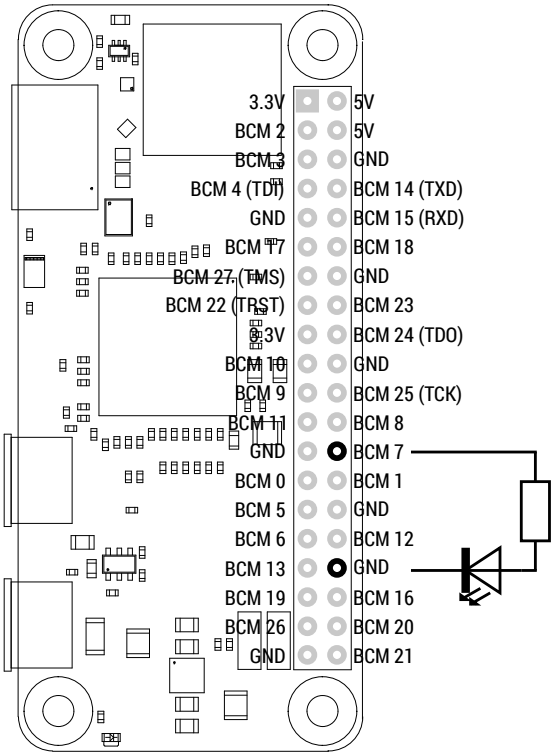

Figure 4.6.: Blinking LED example

the name service. It looks up the requested service (handle to the kernel object), duplicates it, so it does not lose it on returning, and returns the duplicate. The return value is copied to the object space of the caller and a reference to it placed in the value stack.

The user-domain now holds the service for the GPIO-device. It sets the function of the pin to output and switches the value of the pin between on and off in a loop. The `DeviceService` for the GPIO-header writes the value to the appropriate memory location to switch the voltage of the GPIO-pin. An example implementation of the GPIO-header driver can be found in Appendix E.

The GPIO-header is actually a bit special, as it is not possible to enumerate any devices behind the GPIO-headers. First there is no enumeration process defined as no standard exists for GPIO-pins, because it is their purpose to be general purpose. An application could choose to communicate with a device over i2c, spi or other protocols and it is not limited by the pins being bound to a certain purpose.

The driver is also completely free in what interface it presents to the applications, i.e. what methods it defines at the `DeviceService`. It would be wise to define standards for certain device types, so a user application can confidently use a portal to a certain device without the need of inspecting the interface of the portal itself.

4.9.2. File Access

The system is in a state with at least one user domain, the name service, the I/O subsystem running and the virtual file system service ready to be used. Figure 4.7 shows the set-up. A user domain wants to open a file from the file system, read some data, write some data and close the file again.

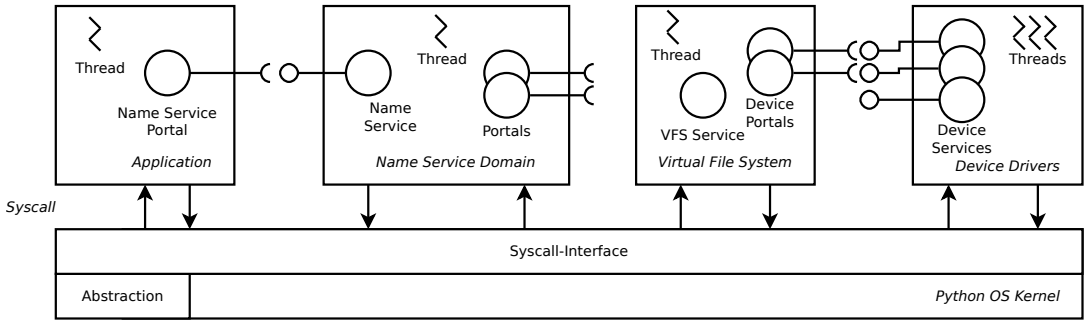

Figure 4.7.: Set-up for the file access example

Assuming the domain was just started, the first action is to ask the name service for a portal to the virtual file system. The virtual file system's portal has been registered at the name service at its start-up. The name service returns the portal of the virtual file system. The user domain uses the `open` method on the portal of the virtual file system to specify the path to the file, it wants to open, and the mode. This instruct the virtual file system to find the file in question.

The virtual file system uses a stack of drivers, namely at least a file system driver (e.g. one for ext4), maybe encryption drivers, and a block device driver for locating the file on the file system. The meta information are read, the permissions of the domain checked and further actions undertaken, which the type of the file system dictate. Assuming the domain has the correct permissions to access the file, a service is created and the portal to it returned to the user domain.

The user domain may read and write data from and into the file, until it decides to close the file portal and destroy the portal. Closing the file (i.e. using the method-call `close` of the file's portal), will instruct the virtual file system to no longer allow any modification from that domain using that specific portal. Assumptions cannot be made however whether or not the modifications to the file have been completed on disk, when the portal call returns. Also the same service may be in use by other domains, accessing the same file.

File portals are also not registered at the name service, as they would need to be updated regularly, i.e. whenever the permissions change, or when a file is deleted. Also permission checking is a process, which should be done by the virtual file system, when the file is requested to be opened, not after the domain already has received a portal. The name service would be bloated if that logic also would need to be included in the name service. Lastly the virtual file system creates file services at run-time, when a file is opened (which exists or was created) and deletes them, when they are no longer used (after all modifications have been written do disk).

File permissions and rights in general where not discussed in this thesis. A similar model to 4.3BSD can be implemented easily, where a new domain inherits the rights (user, group) of the parent. The rights would be noted in a field in the domain control block. The virtual file system would then need a syscall to access this rights-field.

4.9.3. IP Networking

A similar problem to file access is networking. In a UNIX system there are system calls for creating a socket and working with sockets. These don't exist in PylotOS. However a user domain may implement a service object, which allows for creating sockets and using them as expected. The user domain

may choose to create a new service object per created socket, or return handles or integer values, which represent a socket, like in UNIX systems. The latter option would mean, that every domain uses the same service, but passes an integer value with every portal-call, denoting the socket to be used. Access control would be needed at the socket-domain.

When a socket is created, at least a new object is created in the socket domain, which describes its configuration. The configurations describe things like the target address, ports, protocols, etc. A user domain may choose to use a socket portal to communicate with other systems, which do not have a service infrastructure and do not use the service protocol to communicate.

The socket domain uses `DeviceService` objects to communicate with the necessary devices and the network stack layers. The bottom-most layer is the network device itself, which is controlled by a device driver.

5. Further Research

Some interesting aspects of the architecture of PylotOS were left open in this thesis. The power management policy is one such aspect. It is central to the power management facility, but too complex to describe in detail as a side note. A good power management policy may not only be power efficient but also be efficient with CPU time or may enforce maximal performance for some I/O devices, while still retaining a reasonable power efficiency overall.

A virtual file system was hinted on, but never discussed thoroughly. Especially the interplay with boot-up and enumeration of hardware devices, the topic virtual file system and the file systems stack is interesting. Not only is opening and accessing files of concern, but a driver for a block device needs to be up and running. For any access to occur to files partitions have to be recognised and file system drivers and encryption drivers stacked on top of each other.

Re-use of existing code would also be a very interesting research topic, especially in regards to the Linux drivers or rump kernel-drivers. This would however make it necessary to run C-code alongside Python-code inside the Python virtual machine or destroy the abstraction of a Python virtual machine all together. A way to integrate C-code would be to run that in the interpreter context, i.e. on real hardware and export an emulated simpler device to the Python virtual machine, so a Python driver can take control over the simplified device. However a concrete plan would need to be laid out, which is beyond the scope of this thesis.

Other interesting research topics are the evaluation of timing constraints in PylotOS. The timing interrupt through the emulated timer only gives rough timing and does not allow precise timing. The problem is, that the bytecode instruction needs to be executed completely before another domain may be scheduled. By moving timing and domain switching into the interpreter, the timing may become more precise, however the separation between virtual and real hardware is muddied.

Also the `TRAP`-instruction needs to be implemented in a Python-friendly way, not only that the bytecode exists and can be executed, but also the keyword in Python code. Also the existence of the additional instruction should not alter the behaviour of any other bytecode. `TRAP` could probably be based on the code for `CALL_FUNCTION`. Also a `TRAP_RETURN` may need to be introduced, so the interpreter can switch back to user-mode and place a result value on the value stack of the caller.

Other problems include the wrapper library for user level applications. It has been discussed very briefly, that a wrapper library for the domains needs to exists, which wraps syscalls and access to kernel objects neatly, but not work has been done in that regard. Implementing the wrapper library is straightforward, once a syscall interface has been defined. Defining the syscall interface depends on the overall implementation of kernel objects.

DMA was not discussed in this thesis, neither when talking about bulk transactions between the object space and the address based memory, nor when considering hardware, which may allow direct memory access. This is a topic, which can be researched further in the future, when some drivers are working and the overall system architecture is fleshed out.

The service architecture allows the location of a service to be transparent to the application. The service object may live on the same machine, even in the same domain, or it may be on a different machine. Further work may be done by not only implementing the service mechanism, but also work on a protocol to make it truly independent from the location.

The inner workings of the protocol and the network communication then have to use the driver architecture. A trade-off may have to be found, whether or not to try and bring the network driver into the PylotOS kernel for performance reasons and allowing the other domains to communicate with other computers in the network, without the necessity of a dedicated network driver on the system (additionally to whatever lives in the kernel).

Patching Micropython and implementing the HAL-module should be easy and is one of the smaller work packages left open. Micropython needs to be extended to allow several interpreter state structures and the HAL needs to implement functionality to switch between interpreter states, or at least heaps at run-time, as well as thread states, when switching the currently executed thread. Additionally the emulated timer needs to be implemented as well as the timer-flag check in the interpreter loop.

The garbage collection mechanism was not touched in the whole thesis. Garbage collection is needed in Python as the virtual machine only knows an object space and sometimes references to object may be lost, i.e. the object becomes inaccessible. This means, the object is no longer needed. Then the memory of that object can be reclaimed and used for other data. The implementation of that reclaiming needs to be done in the interpreter as part of the garbage collection.

There are loads of garbage collection algorithms from reference counting to mark-and-sweep approaches or other generational garbage collecting approaches. In this thesis garbage collection was assumed to be working, and thus the functionality was not explained. Future research needs to implement different garbage collection algorithms and maybe even make them selectable for the domains at run-time. Performance measurements could be interesting. Also some assumptions could be dropped, like the assumption that no garbage collector may run, when a service call is ongoing. The implications would need researching, when removing assumptions like that. Also in the context of service objects and kernel object this would be very interesting.

Also the keywords `open` and `import` need to be researched closely in regards to the PylotOS operating system, as `import` cannot be used, without a virtual file system or a list of baked in modules to use. `open` is based on using a service in the PylotOS architecture. An application developer needs to keep that in mind. Also the semantics of `import` is not well defined in that context - what does it mean, when a domain used `import`? Which policy decides which modules may be loaded by what domain?

Also further optimisations are possible. Focusing all syscalls around the kernel object-mechanism may make the interface to the kernel clearer. Most physical devices and virtual devices are represented as services, so they already are kernel objects. Mutexes, semaphores and monitors were not described, but will need a generic mutual exclusion implementation in the kernel. Also domains may be referenced as kernel objects.

The signal-concept of Fuchsia has been left out. Signals (flags) for kernel objects may make working with them easier, as it would be possible to query the state of a kernel object, without waiting or explicitely waiting for them, i.e. waiting until a file becomes readable.

6. Conclusions

This thesis described the overall architecture, the machine model and the process architecture, IPC and the driver model of an operating system in Python. It outlined theoretical operating system considerations of Tanenbaum [22] and the practical considerations of different hardware platforms like the x86-platform or the Raspberry Pi. Certain design choices in the CPython interpreter were shown. Based on a literature study of different operating systems like Linux 2.6, Windows 2000, 4.3BSD, JX, JavaOS and Fuchsia, an architecture for the Python OS was developed. The architecture was however not be implemented and evaluated on real hardware.

The mapping between the real machine and the virtual Python machine have been shown in detail. The interpreter as the mediator seeks to broker between the two worlds. It created object spaces for Python from address based memory and handles interrupts of the real hardware by interrupting the virtual machine and executing Python interrupt handlers. The overall system architecture and the separation between the real hardware and virtual machine have been shown.

Although the process abstraction of only 4.3BSD, JX and Fuchsia have been studied, the domain concept of PylotOS is a strong part of the operating system design. It protects processes (domains) of PylotOS against each other in regards to the Python virtual machine. The domains can interact with the kernel and kernel objects via a system call interface, which is implemented in the Python virtual machine by a new bytecode `TRAP`.

The concept of kernel objects of Fuchsia [9] has been adopted to PylotOS, however without the signals (flags) associated to them in Fuchsia. Kernel objects are objects, which share state between domains and allow for transfer of data between domains as well as the destruction of the isolation. They are represented by a numerical value to user space. Modification of kernel objects can be requested with syscalls from user land.

The syscall mechanism was described, with the use of the newly introduced `TRAP`-bytecode. A trap brings the interpreter in kernel state, so kernel code can access all necessary data structures and can access domain's memory and the thread's stacks as well as state-variables in DCBs or Thread Control Blocks (TCBs).

A service / portal mechanism like in JX [8] was introduced on top of these kernel objects and syscalls, which allows an RMI-like access to remote objects across the system. The access to the remote service objects is implemented by so called portals (stubs) to the functionality of the real remote object. These stubs have the same methods as the service object, but map these function calls to interprocess communication instead of just executing code. The majority of the logic needs to be implemented in the user mode wrapper library.

Functionality like the socket library, which is normally found in conventional operating systems may be implemented through services, i.e. a domain offers a service object, which implements sockets. Also a name service was proposed, where user code could register its services with a name for other domains to find. This alleviates the problem of finding other domain's services, which was present in BSD before the introduction of sockets in 4.2BSD.

The driver architecture builds on top of domains and services. The I/O Manager was described in detail, which loads, manages and unloads the drivers in the system. Not only does it start the device enumeration process, but it loads new drivers, when necessary, binds and unbinds devices to and from drivers, and decides over the power policy of the system. It is also the central manager for the plug and play functionality, i.e. detecting new devices at run-time and removing them, when the user requests it or a device was removed surprisingly.

Drivers in the driver model may also use functionality offered by other drivers or device-abstractions. The categorisation of bus, filter and function drivers was adopted from Windows 2000 [21]. The main focal point was accessing hardware in a safe manner from driver domains, without any of the other domains being at risk of attack by a driver. The `Memory` kernel objects ensure, that a driver only accesses memory, which it previously declared as device memory. Other accesses may be invalid and result in raising an exception.

Drivers in PylotOS not only implement access to devices by executing normal user level code, but also may register interrupt handler for catching an interrupt from a device it manages. These interrupt handlers are also written in Python as they handle an exceptional state of the Python virtual machine. DMA was however not discussed in detail.

While key aspects of the driver interface like the virtual file system and power management have been mentioned, they were not fully defined in this thesis. For example no clear interface for certain classes of devices was developed, but merely hinted upon. Clear interfaces need to be developed or adopted from other systems like Fuchsia, to minimize the need for users to use generic calls like `ioctl`.

The majority of this thesis was the literature study of several operating systems. Their designs have influenced the design of PylotOS greatly. The concepts of most conventional operating systems are quite similar, while more unusual systems have more unusual concepts, like Fuchsia, which does not adhere to a POSIXy system call interface. Not all of the concepts introduced in the literature study have been described fully. Also some concepts were not used, but some ideas have been adopted for the PylotOS operating system.

In total, this thesis showed the architecture of a generic interpreted operating system and the problems therein. Solutions have been developed and the separation between real and virtual machine have been shown. The conceptual work in this thesis now has to be implemented in future research.

A. Linux File Operations

Listing A.1: file operations (include/linux/fs.h)

```
struct file {
    struct list_head    f_list;
    struct dentry       *f_dentry;
    struct vfsmount         *f_vfsmnt;
    struct file_operations  *f_op;
    atomic_t        f_count;
    unsigned int        f_flags;
    mode_t          f_mode;
    int         f_error;
    loff_t          f_pos;
    struct fown_struct  f_owner;
    unsigned int        f_uid, f_gid;
    struct file_ra_state    f_ra;

    size_t          f_maxcount;
    unsigned long       f_version;
    void            *f_security;

    /* needed for tty driver, and maybe others */
    void            *private_data;

#ifdef CONFIG_EPOLL
    /* Used by fs/eventpoll.c to link all the hooks to this file */
    struct list_head    f_ep_links;
    spinlock_t      f_ep_lock;
#endif /* #ifdef CONFIG_EPOLL */
    struct address_space    *f_mapping;
};
// ...
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned ↩
        long, unsigned long);
    int (*check_flags)(int);
    int (*dir_notify)(struct file *filp, unsigned long arg);
    int (*flock) (struct file *, int, struct file_lock *);
};
```

B. Windows I/O Request Package Structure

Listing B.1 shows the structure of an IRP in Windows. This definition may be newer than Windows 2000, so it is not listed in Section 3.5

Listing B.1: Windows I/O request package structure from [16]

```
typedef struct _IRP {
  PMDL            MdlAddress;
  ULONG           Flags;
  union {
    struct _IRP  *MasterIrp;
    PVOID       SystemBuffer;
  } AssociatedIrp;
  IO_STATUS_BLOCK IoStatus;
  KPROCESSOR_MODE RequestorMode;
  BOOLEAN         PendingReturned;
  BOOLEAN         Cancel;
  KIRQL           CancelIrql;
  PDRIVER_CANCEL  CancelRoutine;
  PVOID           UserBuffer;
  union {
    struct {
      union {
        KDEVICE_QUEUE_ENTRY DeviceQueueEntry;
        struct {
          PVOID DriverContext[4];
        };
      };
      PETHREAD   Thread;
      LIST_ENTRY ListEntry;
    } Overlay;
  } Tail;
} IRP, *PIRP;
```

C. CPython Opcodes

Listing C.1: List of known opcodes in CPython, *Include/opcode.h* [6]

```
/** ... **/
/* Instruction opcodes for compiled code */
#define POP_TOP                   1
#define ROT_TWO                   2
#define ROT_THREE                 3
#define DUP_TOP                   4
#define DUP_TOP_TWO               5
#define ROT_FOUR                  6
#define NOP                       9
#define UNARY_POSITIVE           10
#define UNARY_NEGATIVE           11
#define UNARY_NOT                12
#define UNARY_INVERT             15
#define BINARY_MATRIX_MULTIPLY   16
#define INPLACE_MATRIX_MULTIPLY  17
#define BINARY_POWER             19
#define BINARY_MULTIPLY          20
#define BINARY_MODULO            22
#define BINARY_ADD               23
#define BINARY_SUBTRACT          24
#define BINARY_SUBSCR            25
#define BINARY_FLOOR_DIVIDE      26
#define BINARY_TRUE_DIVIDE       27
#define INPLACE_FLOOR_DIVIDE     28
#define INPLACE_TRUE_DIVIDE      29
#define GET_AITER                50
#define GET_ANEXT                51
#define BEFORE_ASYNC_WITH        52
#define BEGIN_FINALLY            53
#define END_ASYNC_FOR            54
#define INPLACE_ADD              55
#define INPLACE_SUBTRACT         56
#define INPLACE_MULTIPLY         57
#define INPLACE_MODULO           59
#define STORE_SUBSCR             60
#define DELETE_SUBSCR            61
#define BINARY_LSHIFT            62
#define BINARY_RSHIFT            63
#define BINARY_AND               64
#define BINARY_XOR               65
#define BINARY_OR                66
#define INPLACE_POWER            67
#define GET_ITER                 68
#define GET_YIELD_FROM_ITER      69
#define PRINT_EXPR               70
#define LOAD_BUILD_CLASS         71
#define YIELD_FROM               72
#define GET_AWAITABLE            73
#define INPLACE_LSHIFT           75
#define INPLACE_RSHIFT           76
#define INPLACE_AND              77
#define INPLACE_XOR              78
#define INPLACE_OR               79
#define WITH_CLEANUP_START       81
#define WITH_CLEANUP_FINISH      82
#define RETURN_VALUE             83
#define IMPORT_STAR              84
#define SETUP_ANNOTATIONS        85
#define YIELD_VALUE              86
#define POP_BLOCK                87
#define END_FINALLY              88
#define POP_EXCEPT               89
#define HAVE_ARGUMENT            90
#define STORE_NAME               90
#define DELETE_NAME              91
#define UNPACK_SEQUENCE          92
#define FOR_ITER                 93
#define UNPACK_EX                94
#define STORE_ATTR               95
#define DELETE_ATTR              96
#define STORE_GLOBAL             97
#define DELETE_GLOBAL            98
#define LOAD_CONST              100
#define LOAD_NAME               101
#define BUILD_TUPLE             102
#define BUILD_LIST              103
#define BUILD_SET               104
#define BUILD_MAP               105
#define LOAD_ATTR               106
#define COMPARE_OP              107
#define IMPORT_NAME             108
#define IMPORT_FROM             109
#define JUMP_FORWARD            110
#define JUMP_IF_FALSE_OR_POP    111
#define JUMP_IF_TRUE_OR_POP     112
#define JUMP_ABSOLUTE           113
#define POP_JUMP_IF_FALSE       114
#define POP_JUMP_IF_TRUE        115
#define LOAD_GLOBAL             116
#define SETUP_FINALLY           122
#define LOAD_FAST               124
#define STORE_FAST              125
#define DELETE_FAST             126
#define RAISE_VARARGS           130
#define CALL_FUNCTION           131
#define MAKE_FUNCTION           132
#define BUILD_SLICE             133
#define LOAD_CLOSURE            135
#define LOAD_DEREF              136
#define STORE_DEREF             137
#define DELETE_DEREF            138
#define CALL_FUNCTION_KW        141
#define CALL_FUNCTION_EX        142
#define SETUP_WITH              143
#define EXTENDED_ARG            144
#define LIST_APPEND             145
#define SET_ADD                 146
#define MAP_ADD                 147
#define LOAD_CLASSDEREF         148
#define BUILD_LIST_UNPACK       149
#define BUILD_MAP_UNPACK        150
#define BUILD_MAP_UNPACK_WITH_CALL 151
#define BUILD_TUPLE_UNPACK      152
#define BUILD_SET_UNPACK        153
#define SETUP_ASYNC_WITH        154
#define FORMAT_VALUE            155
#define BUILD_CONST_KEY_MAP     156
```

```
#define BUILD_STRING            157
#define BUILD_TUPLE_UNPACK_WITH_CALL 158
#define LOAD_METHOD             160
#define CALL_METHOD             161
#define CALL_FINALLY            162
#define POP_FINALLY             163

/* EXCEPT_HANDLER is a special, implicit
   block type which is created when
   entering an except handler. It is not an
   opcode but we define it here
   as we want it to be available to both
   frameobject.c and ceval.c, while
   remaining private.*/
#define EXCEPT_HANDLER 257
/** .. **/
```

D. Prototypic Implementation of the Service User Mode Wrapper

Listing D.1: Prototypic implementation of the service class in the user mode wrapper library

```
## @package portal

import sys
import inspect
import socket
import pyos

class service:
    def __init__ ( self ):
        self.domainId = pyos.getDomainId();
        self.handle = pyos.registerServiceObject ( self.listMethods() );
        pass

    ## @brief create a list of methods of the self-object, including the names of the ↩
        parameters
    def listMethods( self ):
        serviceMethods = [func for func in dir(service) if callable(getattr(self, func))]
        methods = [func for func in dir(self) if callable(getattr(self, func)) and func not ↩
            in serviceMethods] # and not func.startswith("__")
        mlist = []
        for fnName in methods:
            codeArgList = ""
            argSpec = inspect.getargspec ( getattr(self, fnName ) )
            argNum = 0
            argTuple=""
            for arg in argSpec.args:
                if ( len(codeArgList) > 0 ):
                    codeArgList = codeArgList + ", "
                codeArgList = codeArgList + arg
                if ( arg != "self" ):
                    argTuple = argTuple + "            '"+arg+"': "+arg+", \n";
                argNum = argNum + 1
            mlist.append ( {"fnName": fnName, "arg": ( codeArgList ) } )
        return ( mlist );

    ## @brief waits on the message queue for a service until a request is present
    def wait ( self ):
        request = pyos.waitForService ( self.handle );
        if ( hasattr ( self, request["fnName"] ) )
            fn = getattr ( self, request["fnName"] )
            args = (self,) + request["args"]
            pyos.returnFromService ( self.handle, fn(*args) )
```

E. Prototypic Implementation of a PylotOS GPIO-driver for Raspberry Pi 1 and Zero

Listing E.1: Prototype implementation of a GPIO-driver in PylotOS for the Raspberry Pi 1 and Zero

```
from userland_memory import *
import iomgr

class gpioHeader ( object ):
    fn5    = iomgr.IORegister (  0 )
    fn4    = iomgr.IORegister (  4 )
    fn3    = iomgr.IORegister (  8 )
    fn2    = iomgr.IORegister ( 12 )
    fn1    = iomgr.IORegister ( 16 )
    fn0    = iomgr.IORegister ( 20 )

    set1   = iomgr.IORegister ( 28 )
    set0   = iomgr.IORegister ( 32 )
    clear1 = iomgr.IORegister ( 40 )
    clear0 = iomgr.IORegister ( 44 )

    lvl1   = iomgr.IORegister ( 52 )
    lvl2   = iomgr.IORegister ( 56 )

    fn       = ( fn0, fn1, fn2, fn3, fn4, fn5 )
    setPin   = ( set0, set1 )
    clearPin = ( clear0, clear1 )
    lvl      = ( lvl0, lvl1 )

    # skipping the rest of the GPIO-registers

    ## set the base address and initialize the memory region
    def __init__ ( self, base ):
        self.baseAddr = base
        self.memory = Memory ( self.baseAddr, 156 )

    ## set the value of a pin (HIGH = 1, LOW != 1)
    def setPin ( self, pin, value ):
        offset = 0
        if ( pin > 32 ):
            pin = pin - 32
            offset = 1

        if ( value == 1 ):
            self.setPin[offset] = self.setPin[offset] | (1 << pin)
        else:
            self.clearPin[offset] = self.clearPin[offset] | (1 << pin)

    ## read the value of a pin
    def getPin ( self, pin ):
        offset = 0
        if ( pin > 32 ):
            pin = pin-32
            offset = 1

        if ( self.lvl[offset] & (1 << pin) != 0 ):
            return 1
        else:
            return 0
```

```
55
56      ## select a function of the GPIO-pin
57      def fnSelect ( self, pin, function ):
58          if ( function > 7 ):
59              return
60          if ( pin > 53 ):
61              return
62
63          offset = 0
64          while ( pin >= 10 ):
65              offset = offset + 1
66              pin = pin - 10
67
68          val = self.fn[offset]
69          val = val & ~(7 << ((pin<<1) + pin))
70          val = val | (function << ((pin<<1) + pin))
71          self.fn[offset] = val
72
73  ## hooks for the user application to use
74  class gpioDevService ( iomgr.DeviceService ):
75      def __init__ ( self, hdr ):
76          self.gpiohdr = hdr
77
78      def setPin ( self, pin, value ):
79          self.gpiohdr.setPin ( pin, value )
80
81      def getPin ( self, pin ):
82          return self.gpiohdr.getPin ( pin )
83
84      def fnSelect ( self, pin, fn ):
85          self.gpiohdr.fnSelect ( pin, fn )
86
87  ## hooks for the I/O Manager to use
88  class gpioService ( iomgr.DriverService ):
89      devSrvcs = []
90      def __init__ ( self, hdr ):
91          pass
92      def enumerate ( self, dev ):
93          pass
94      def power ( self, dev ):
95          pass
96      def bind ( self, identifier ):
97          gpiohdr = gpioHeader ( 0x20000000 + 0x200000 )
98          devService = gpioDevService ( gpiohdr )
99          devSrvcs.append ( devService )
100         return devService
101     def unind ( self, devService )
102         # find the correct devService and delete it
103         pass
104     def release ( self ):
105         pass
106     def softexc ( self, irqPackage ):
107         pass
108
109 iomgr.portal.register ( gpioService () )
```

F. Prototypic Implementation of the generic Python Interrupt Handler

Listing F.1: Prototypic implementation of the generic I/O exception handler

```
# map out the interrupt vectors of the processors
pyos_irq_reason = [
    pyos_reset,
    pyos_undef,
    pios_swi,
    pyos_abort,
    pyos_abort,
    pyos_reset,
    pyos_irq,
    pyos_fiq
];

# manage a list of interrupt handlers according to the line of their device
pyos_irq_lines = { 42: [exampleExceptionHandler] }

# manage a list of syscalls
pyos_irq_syscalls = {
    1: pyos_syscall_service_call,
    2: pyos_syscall_service_return,
    # ...
}

def pyos_reset ():
    pass
# ...

# either use software interrupt for syscall or alter the interpreter to
# set a different reason
def pyos_swi ():
    number = pyos.getTrapNumbers ();

    # get the topmost element of the value stack - parameters to the syscall
    parameters = pyos.popValueStack ( pyos.currentlyRunningThread );

    if ( callable(pyos_irq_syscalls[number]) ):
        val = pyos_irq_syscalls[number] ( parameters )

    # push the return value back to the value stack of the caller thread
    pyos.pushValueStack ( pyos.currentlyRunningThread, val );
    return 0;

## find out the raised irq line and call the appropriate handler(s)
# @return 0 on success, something else otherwise
def pyos_irq ():
    line = hal.raisedLine ()
    try:
        if ( type(pyos_irq_lines[line]) is list ):
            for h in pyos_irq_lines[line]:
                if ( callable(h) ):
                    v = h()
                if ( v == 0 ):
                    return 0;
    finally:
        pyos_syslog ( "No suitable irq handler found for line %d"%(line) )
        return 42

## handle any interrupt to the processor or the virtual machine
```

```
58 # @param reason the reason of the interrupt, the first level C-routines
59 #               call this generic handler with the reason as numeric argument
60 def pyos_generic_irq_handler ( reason ):
61     if ( reason > len(pyos_irq_reason) ):
62         pyos_syslog ( "Unknown Interrupt Reason %d"%(reason) );
63         return;
64 
65     val = 1;
66     try:
67         if ( callable ( pyos_irq_reason[reason] )) :
68             val = pyos_irq_reason[reason]()
69     except:
70         pass
71 
72     if ( val != 0 ):
73         pyos_syslog ( "Could not handle interrupt %d"%(reason) );
74 
75     return;
```

Declaration of authorship

Ich erkläre an Eides statt, gegenüber der Technischen Universität Chemnitz, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angebenen Quellen und Hilfsmittel angefertigt habe. Diese Aussage gilt auch für die Implementation und Dokumentation im Rahmen diesen Projekts.

I hereby certify to the Chemnitz University of Technology that this thesis is completely my own work and uses no external material other than that acknowledged in the text. This declaration also holds for the implementation and documentation done in the context of this project.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche gekennzeichnet.

This work contains no plagiarism. All sentences or passages directly quoted from other people's work or content derived from such work have been specifically credited to the authors and sources.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keinem anderen Prüfer als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

This thesis has neither been submitted in the same or a similar form to any other examiner nor for the award of any other degree nor has it previously been published.

Chemnitz, 11th July 2018 (Naumann, Stefan)

Bibliography

[1] Yaniv Aknin. Pythons Innards: Introduction. https://tech.blog.aknin.name/2010/04/02/pythons-innards-introduction/. [accessed 2018-04-30].

[2] ARM. ARM1176jzf-s Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/DDI0301H_arm1176jzfs_r0p7_trm.pdf. [accessed 2017-09-20].

[3] Daniel Pierre Bovet and Marco Cesati. *Understanding the Linux kernel : [from I0 ports to process management; covers version 2.6]*. O'Reilly, Beijing, 3. edition, 2006.

[4] Broadcom. BCM2838 ARM Peripherals Manual. https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf. [accessed 2017-09-20].

[5] Sean M Dorward, Rob Pike, David Leo Presotto, Dennis M Ritchie, Howard W Trickey, and Philip Winterbottom. The inferno operating system. *Bell Labs Technical Journal*, 2(1):5–18, 1997.

[6] Python Software Foundation. Opcodes in CPython (*Include/opcode.h*. https://github.com/python/cpython. [accessed 2017-07-05].

[7] Damien George and Stefan Naumann. Github-issue: Wrong label-calculation for JUMP-Bytecode on Raspberry Pi, due to unaligned access. https://github.com/micropython/micropython/issues/3201. [accessed 2017-11-16].

[8] Michael Golm. *The structure of a type safe operating system* . PhD thesis, Zugl.: Erlangen, Nürnberg, Univ., Diss., 2003., Erlangen, 2003.

[9] Google. Fuchsia is not Linux. https://fuchsia.googlesource.com/docs/+/HEAD/the-book/README.md. [accessed 2018-04-25].

[10] Thomas Hallgren, Mark P Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in haskell. In *ACM SIGPLAN Notices*, volume 40, pages 116–128. ACM, 2005.

[11] Peter Hinch and Mike Causer. Writing Interrupt Handlers - Micropython Documentation. http://docs.micropython.org/en/latest/pyboard/reference/isr_rules.html. [accessed 2017-09-12].

[12] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007.

[13] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Qartermann. *The design and implementation of the 4.3BSD UNIX operating system*. Addison-Wesley, Reading, Mass., reprinted with corr. edition, 1990.

[14] Cláudio Maia, Luis Miguel Nogueira, and Luis Miguel Pinho. Evaluating android os for embedded real-time systems. In *6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 63–70, 2010.

[15] Microsoft. Device Tree | Microsoft Docs. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/device-tree. [accessed 2017-09-25].

[16] Microsoft. IRP structure (Windows Drivers). https://msdn.microsoft.com/en-us/library/windows/hardware/ff550694(v=vs.85).aspx. [accessed 2017-09-22].

[17] Stefan Naumann. Adding a module to Micropython. https://www.stefannaumann.de/en/2017/07/adding-a-module-to-micropython/. [accessed 2017-07-24].

[18] J. M. O'Connor and M. Tremblay. picojava-i: the java virtual machine in hardware. *IEEE Micro*, 17(2):45–53, Mar 1997.

[19] redhat. The Red Hat newlib C Library. http://sourceware.org/newlib/libc.html#Syscalls. [accessed 2017-08-08].

[20] S. Ritchie. Systems programming in java. *IEEE Micro*, 17(3):30–35, May 1997.

[21] David A. Solomon and Mark E. Russinovich. *Inside Microsoft Windows 2000* . Microsoft Press, Redmond, Wash., 3. ed. edition, 2000.

[22] Andrew S. Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, Boston, Mass., 4. ed., global ed. edition, 2015.

[23] P. Tröger, C. Jakobs, T. Jakobs, and M. Werner. Adaptive cyber-physical systems with interpreted operating system kernels. In *2016 5th Mediterranean Conference on Embedded Computing (MECO)*, pages 26–29, June 2016.

Glossary

Block Stack in Python keeps information about the current code block, e.g. exception handlers, or loops. See Section 4.1.1.

Call Stack in Python keeps information about the code objects, which are currently being evaluated. The call stack consists of frame objects, which are contain references to the other stacks: the value stack and the block stack.

Domain in PylotOS is the unit of protection. User code runs in domains without the ability of accessing any data of other domains. Domains can interact with kernel objects and the PylotOS kernel via system calls.

Domain Control Block container for all relevant information about a domain. Contains identificator, object space information, table of handles and interpreter state.

Kernel Object are privileged objects, which reside in the kernel space. They are not directly accessible by any user code, but an action can be requested via system calls. `Memory` and `Services` are kernel objects.

Object Space in PylotOS is a memory region, the interpreter uses to allocate, store, modify and delete Python objects. The memory is transparent to user code. Object spaces are garbage collected.

Service/Portal in PylotOS exports functionality to different domains. A portal is a remote reference to a service object in a different domain. Designed after the RMI-principle, i.e. it is transparent to a domain, where the called service object resides. It could reside in the same or a different domain and maybe even different machine.

System Call A call to the operating system requesting certain actions. Normally hidden from the programmer by library calls. In PylotOS implemented via the new `TRAP`-instruction.

Thread Control Block container for relevant information for a thread. Contains thread state, thread identificator.

Value Stack (or evaluation stack) in Python is the place, where objects are manipulated, when certain opcodes are evaluated, i.e. some opcodes work directly on arguments on the value stack.