University of Warsaw
Faculty of Mathematics, Computer Science and Mechanics

Marek Dzikiewicz
Student no. 234040

Input/Output Subsystem in Singularity Operating System

Master's Thesis
in COMPUTER SCIENCE

Supervisor:

Janina Mincer-Daszkiewicz, Ph.D.
Institute of Computer Science,
University of Warsaw

June 2011

Abstract

Microsoft Singularity is a research operating system, which explores the possibilities of designing and building software platforms for reliable computer systems. Singularity uses processes to isolate system services and device drivers from the rest of the system, and to enable restarting failed components without affecting the entire operating system. This leads to better support for recovering from errors in system components and makes the system more dependable.

The original version of Singularity does not support executing bus device drivers in separate processes and implements them as part of the operating system's kernel. This thesis proposes a new bus driver model which extends the original implementation of Singularity and supports process-isolated bus drivers. The main concepts behind Singularity are briefly described and Singularity's input/output subsystem is presented in-depth.

Keywords

Singularity, Microsoft, software isolated process, driver, managed code

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

D. Software
D.4. Operating Systems
D.4.4. Communications Management

Tytuł pracy po polsku

Podsystem wejścia-wyjścia systemu operacyjnego Singularity

Contents

Introduction

Operating systems are one of the most essential elements of all computer systems. Their stability and functionality affects all applications, from home and entertainment programs, through business solutions and finally to real-time systems, on which even human life may depend.

It seems that operating systems should be one of the most dynamically developed types of software. However, in reality, the architecture of most contemporary operating systems was designed in the 1960's and 70's. The reason for this fact is that every major modification in an operating system usually causes loss of compatibility with applications which were written for that system. Therefore rebuilding an operating system would require rebuilding the entire software which the system's users utilize. That is why modern operating systems are actually systems from many years ago, which have been extended and adapted to modern applications. Nevertheless, the prevalence and application of computer systems in the 70's was completely different than today.

Nowadays, one of the most important requirements for computer systems is dependability. The increase in available hardware resources resulted in the fact, that even performance is not as important as a well defined operation of systems according to their specification. In many situations, computer solutions are not yet used, because of the lack of trust for software, which acts unpredictably and causes many problems. However, it is extremely difficult to build stable computer solutions based on system platforms from many years ago, which were not designed for modern applications. No wonder that contemporary computer systems are unstable and unreliable.

In 2003, Microsoft Research started working on a project called "Singularity" [5]. The aim of the project was to investigate possibilities for development of dependable computer systems. One of the project's outcomes was an experimental, prototype operating system called "Singularity". The system was designed and written from scratch, using modern programming languages, and based on assumptions and architectures which satisfy contemporary needs.

One of the main concepts of Singularity is component isolation. The system is divided into small, independent components and each component is executed in a separate process. Thanks to this separation, in case of a failure in one of the components, it is not necessary to restart the entire system, but only the single component which failed. Particularly device drivers, which are very unreliable software [7], are executed in separate processes and isolated from the rest of the operating system.

However, not all Singularity device drivers run in separate processes. The original version of Singularity's input/output subsystem contains bus device drivers which are implemented as part of Singularity's kernel. Therefore an error in one of these drivers causes a fatal error of the entire system. This raises the probability of a system failure and decreases its reliability.

The main aim of this thesis is to design and implement a bus device driver model, which supports process-isolated bus drivers. Such implementation will improve Singularity's reliability by isolating bus driver failures from the system. It will also make the implementation of Singularity's drivers more consistent. Furthermore, such a driver model will be very useful when drivers for complex bus devices, such as the USB bus, will be developed in the future.

However, creating such a process-based bus driver model for Singularity is not an easy task. First of all, there is very little documentation on Singularity's I/O subsystem, so most of the design and behavior of the subsystem will have to be deducted from source code. Moreover, Singularity is implemented in research programming languages with many new language constructs, which need to be familiarized. Furthermore, Singularity itself implements many new concepts, which are not easy to understand and the new driver model will need to use most of these concepts.

The first chapter describes the main concepts of Singularity, its assumptions, and the system's architecture. All topics in this chapter have been described briefly, but deeply enough for the reader to understand the rest of the thesis. The second chapter describes the original implementation of Singularity's input/output subsystem. The next chapter presents how the original implementation of that subsystem can be extended to support process-isolated bus device drivers. The chapter introduces a newly designed bus driver model, which extends Singularity's standard driver model and enables executing bus device drivers in separate processes, outside the system's kernel. Additionally, tests of the implementation of this new driver model are presented at the end of the chapter. The last chapter concludes the thesis.

Chapter 1

Singularity Operating System

Singularity is a quite an unconventional operating system. As a research project, its main purpose is to try out new ideas for building operating systems. Therefore Singularity implements many features not found in contemporary operating systems.

This chapter presents the most important concepts behind Singularity and briefly describes the system's architecture. The purpose of this part of the thesis is to introduce the reader to ideas which will be referenced later on. A full description of Singularity's concepts is, however, out of the scope of this thesis and only the most important aspects are presented. For a more detailed explanation of the topics from this chapter, please see [5].

1.1. Programming Languages for Singularity

Singularity is based on modern, type-safe programming languages, tools and compilers. One of the main concepts of the project is to use the knowledge acquired during development of modern, high-level programming languages and tools, to build reliable operating systems.

The following section explains what safe programming languages are and introduces the languages which are used to implement applications for Singularity, as well as the operating system itself.

1.1.1. High-level and Safe Programming Languages

Most modern computer systems are implemented using high-level programming languages. The reason for this is that high-level programming languages have many advantages, which simplify designing, implementing and maintaining large software systems.

First of all, high-level programming languages provide rich, object-oriented language features (such as classes, inheritance, polymorphism, etc.) which make it possible to build large systems out of many independent components. Such components encapsulate implementation details and enable code reusability. This way programmers can build new components out of smaller ones, writing a high-level code which focuses rather on the application's logic than the low-level implementation details. Systems which are built in this manner are easy to understand, analyze, test and therefore maintain. Of course lower-level programming languages, such as C or C++, also enable such a programming style, however high-level languages like C# or Java are oriented at such a programming style and support a wide range of programming constructs which simplify building large software systems.

One of the most significant advantages of high-level programming languages is the possibility to express precise interfaces between components and the usage of these interfaces is validated at compile-time. This property of programming languages is usually called type-safety. For example, in type-safe languages, it is possible to define that a list of integer values is a component which enables adding, removing and enumerating through a collection of integer values. However, such a component can be used to store integer values only, and an application which tries to store a different value on such list will not be compiled successfully. Furthermore, defining precise types of all objects used by the programmer allows the compiler to detect a lot of programming errors at compile-time. That is very important when it comes to building large systems, because it allows to automatically detect errors in code as early as possible – that is during compilation. Using low-level languages, such as C, often leads to detecting errors at run-time, which is usually complicated and time-consuming. While languages such as C are type-safe to some extent, component-based languages, such as C# and Java, take type-safety to a whole new level by introducing mechanisms such as generic types (types parameterized by other types [8]), delegates (type-safe equivalent of C function pointer), etc. Programming languages developed during the Singularity project extend languages such as C# even further, enabling compile-time verification of many more aspects and enabling the programmer to express many additional constraints on components.

Another major advantage of high-level programming languages is automatic memory management, which provides automatic memory allocation and deallocation by the runtime environment instead of requiring the programmer to write explicit system calls. Programming languages which implement such mechanism are called managed languages. Memory management is very complicated and very hard to implement properly. Software written in unmanaged languages, such as C or C++, usually contains many hard to fix errors, such as memory leaks, invalid memory references, etc. Furthermore, such applications are susceptible to security vulnerabilities, such as buffer overflow errors, which in case of operating systems are extremely dangerous and compromise the system's security. Managed languages never implement mechanisms which allow direct memory access (such as pointers in C). Thanks to this, the programmer is not even able to reference invalid memory or leak memory. However, the price for automatic memory management is a decrease in performance, due to the overhead of the language runtime.

In spite of all advantages of high-level programming languages, operating systems are usually written in low-level languages, such as C. One of the reasons is that in the past, when most popular operating systems were developed, high-level programming languages were still being developed and the ones that existed back then were not robust enough. Furthermore, because of the low-level nature of operating systems, it is sometimes easier to program them in low-level languages. Mechanisms like memory management, device communication or resource scheduling is easier to express in low-level languages, especially if the implementation must be very efficient (which is crucial in the case of operating systems).

Singularity is based on safe programming languages. Such languages must ensure type safety and memory safety. Type safe languages guarantee that objects and values are always properly interpreted and manipulated. Memory safety guarantees that only valid memory regions (belonging to the calling process and representing live objects managed by the runtime) can be referenced. C# and Java are examples of modern, safe programming languages.

Members of the Singularity project decided that reliable software must be based on safe programming languages. That's why in the case of Singularity, not only the operating system itself is written in high-level, managed languages[1], but also all software executed

[1]A small part of Singularity's kernel is written in unmanaged languages (namely C++ and x86 assembler), due to their low-level nature.

on Singularity, including device drivers, must be written in safe languages. This means that on Singularity, it is not possible to run software compiled to 32-bit or 64-bit machine code. Software for Singularity must be written in a high-level programming language and presented to the system as MSIL[2] code. This guarantees that only a type safe and memory safe code will be executed on the system. The MSIL code is not executed directly on a virtual machine – it is translated into binary code by so called code translators, which are a part of the system's trusted computing base (TCB). Singularity controls the process of code translation, so the generated binary code can be highly optimized. The result of the code translation is digitally signed and cannot be modified later.

The idea of making an operating system accept only MSIL code, has very serious consequences. Almost all software which is currently used, must be rewritten in new programming languages, in order to execute them on Singularity. This would be unacceptable in the case of a regular, commercial, commonly used operating systems like Windows or Linux because of the lack of backward compatibility. However this is not the case of Singularity, which is a research operating system and does not have to be compatible with anything.

Because all applications for Singularity are compiled to MSIL code, it is very easy to safely merge applications from components written in many different programming languages (assuming that they are safe languages). Applications for Singularity are usually written in one of three languages. First of them is the regular C# (the same one which is used to create applications for the .NET Framework on Windows and the Mono platform on Linux). The second language is Spec# [1], which is C# extended with language constructs which enable defining additional assertions and conditions, which are validated at compile-time and run-time. The third language is Sing# [2], which is Spec# extended with language constructs specific for the Singularity operating system.

1.1.2. Code Verification

Singularity extensively uses static code verification mechanisms, to make sure that applications work properly in the system. Thanks to the use of MSIL as Singularity's binary executable format, a lot of errors, such as invalid pointer arithmetic, are detected at compile time. Together with the assumptions of Singularity's process model (described later in this thesis), the system can verify whether an application is compatible with its execution environment, even before it is started.

Although compilers for safe languages are able to detect a lot of errors at compile-time, some errors cannot be detected at this phase (for example a library required by the application may not be installed). In most operating systems, such errors are handled at run-time, after the application has been started. There are two problems associated with this solution. Firstly, an application which causes an error at runtime may endanger the stability of the operating system, as well as other applications (for example, a plugin extension may cause its host application to fail). Secondly, run-time errors may be very difficult to find and analyze. The earlier an error is detected, the better.

That's why Singularity introduces two additional phases of error detection, which occur between compilation and run-time. The first one is called the installation phase. In contrast to standard operating systems, in which application installation is usually restricted to copying files, in Singularity application installation is a strictly defined process. It includes saving all data required to run the application and registration of the application in the system. During installation the application's MSIL code is verified and compiled to machine code.

[2]Microsoft Intermediate Language (MSIL) is the native language of the virtual machine used by Microsoft .NET Framework.

Then, the generated code is digitally signed by the system, to ensure that it is not modified later. During the installation phase, Singularity can detect a lot of errors, which in standard operating systems would be found only at run-time. For example, the system can make sure that all necessary components are installed, so an application will never be started without dependent libraries. Furthermore, Singularity can make sure that installation of the new application components will not break applications which are already installed in the system.

The second phase introduced by Singularity is called the load-time phase. It occurs when the application is loaded into the system, before its execution. The main purpose of this phase is to check whether the application is compatible with the current system environment. For example, Singularity may detect that a service required by the application is not running. This minimizes the probability of application failure because of execution in an unsupported environment. The load-time phase is very important in the case of device drivers, because the system can guarantee that device drivers use only devices for which they are installed, and that conflicting drivers are never executed simultaneously.

Thanks to MSIL code and the two additional error detection phases, Singularity is able to detect errors as early as possible and minimize the amount of run-time errors, which decrease the system's stability and in case of device drivers, may even cause a system-wide failure.

1.2. Software Isolated Processes

Errors occur in every computer system and the bigger the system, the greater the probability of a failure. A good operating system has to be ready to handle situations, in which applications fail and perform illegal operations which are inconsistent with their specification. The problem is that most operating systems do not handle application failures well, especially in the case of system applications and device drivers – their failure usually causes a failure of the whole system. The main reason for this fact is that system services and device drivers are usually a part of the system's kernel and they are not isolated from the operating system (and other applications). Therefore it is very hard to stop or restart system services or device drivers without affecting the system's stability.

The designers of Singularity addressed this problem by introducing mechanisms for isolating software components as part of the system's architecture. The most important mechanism is called software isolated processes (abbreviated SIPs). This section describes the key ideas behind this mechanism.

Software isolated processes are Singularity's equivalent of processes from standard operating systems. SIPs establish a context for the code of executed programs and they are an abstraction for a single instance of an executed program. Furthermore, SIPs isolate programs from each other and they are used to schedule resources, such as CPU, memory, etc.

1.2.1. Closed Object Spaces

From the programmer's point of view, processes in Singularity are object spaces, in contrast to standard operating systems, where processes are actually memory spaces. In contemporary operating systems, applications operate on memory directly, using machine code instructions. In Singularity, processes can only operate on objects using MSIL code, so objects are the only abstraction of memory in the system. The most important fact is that in Singularity, applications cannot reference memory directly. Instead, they must use typed object references. Together with garbage collection mechanisms, this eliminates the problem of dangling pointers and referencing uninitialized memory. Furthermore, it is easy to protect system code from being executed by applications outside the kernel, since classes without public constructors cannot be instantiated (and therefore used) by non-system applications. Similarly, SIPs cannot access memory regions, to which they have no references.

There is one more substantial difference between SIPs and traditional processes. After a SIP is loaded into memory and starts its execution, no additional code can be loaded into the SIP's code space and no code can be modified. In other words, the entire code must be specified before the SIP begins execution. This solution was implemented in Singularity primarily for two reasons. First of all, closed code spaces make it possible to use better static code analysis and optimization algorithms. Secondly, the closed code space concept supports writing applications as sets of many small, well-isolated components. Modern applications very often use dynamic code loading as a mechanism of providing extensibility. Components which extend the application's functionality are dynamically loaded and become part of the application. The problem with such architecture is error propagation – an error in one of the components usually leaves the application in an unknown state and requires restarting the whole application. In Singularity extensions must be loaded into the system as a separate SIP which communicates with the rest of the application's processes. In case of an error, it is usually possible and sufficient to restart only the SIP of the extension which failed.

1.2.2. Hardware and Software Memory Protection

One of the most important tasks of an operating system is isolating processes, to enable simultaneous execution of many programs. Particularly, an operating system must guarantee that no process can read or write memory which belongs to another process. Usually such a functionality is implemented using hardware memory protection mechanisms. When a process tries to access a memory address which is invalid, or points to a memory region that does not belong to the process, then the operating system is signalled about the situation by a hardware interrupt. In this case, the process which executed the illegal operation is usually terminated by the operating system. Hardware memory protection is very powerful and quite easy to implement, however it has a major cost in terms of performance.

In contrast to standard processes, software isolated processes do not rely on hardware to provide process isolation. This is possible only thanks to the fact, that programs in Singularity are expressed as MSIL, which does not enable direct memory referencing – memory can be accessed only by object references, which always point to valid memory regions belonging to the process. The programmer simply cannot write a code which references invalid memory. This leads to the conclusion, that in Singularity it is possible to simultaneously run multiple processes without hardware memory protection and this gives many advantages. First of all processes can be started and stopped a lot faster then in traditional operating systems, because the hardware protection mechanisms do not have to be configured and managed for each process. Furthermore, the memory footprint of processes is a lot smaller without hardware memory protection and this enables faster context switching of processes and threads, which is very important in applications which use multi-threading and parallel computations.

1.2.3. The State Sharing Problem

One of the major problems associated with providing isolation between processes is state sharing. Every operating system which supports multitasking must provide some sort of mechanism for interprocess communication. Usually this mechanism is implemented as sharing a region of memory which many applications can read and write. For programmers this is the most intuitive way to share data between many processes. At the same time this mechanism is extremely difficult to use properly and many programmers do not realize that. For this reason applications which use shared memory usually contain many errors which are very hard to find and fix.

The described memory sharing mechanism has even a more serious flaw – it leads to state sharing. When an application, which uses shared memory, fails due to an error, then the data stored in the shared memory region is in an unknown state. It might have been corrupted by the application which failed. This leads to the necessity to terminate all applications which use the shared memory region, which in turn results in serious error propagation between processes in the system. This situation is even more dramatic when it comes to system components whose errors can easily cause a failure of the entire operating system.

Due to the presented reasons, Singularity does not allow memory sharing. The architecture of the system makes it impossible to share objects between processes or the system's kernel. This has many advantages, for example each process can have a different runtime environment and use a different garbage collection algorithm. Furthermore, it is easier to reclaim memory, after a process is terminated. On the other hand, interprocess communication is more complex and requires special system mechanisms.

1.3. Interprocess Communication in Singularity

Building applications for Singularity often involves dividing them into many well-isolated SIPs, due to the fact that Singularity does not allow dynamic code loading. All components which are added to the application as extensions or plug-ins must be implemented as separate processes. Furthermore, software isolated processes cannot share memory. Therefore, there must exist some sort of mechanism which would allow to exchange data between processes. That's why Singularity implements communication channels – a reliable and robust interprocess communication mechanism.

Singularity's communication channels are described briefly in this section. For more details on interprocess communication in Singularity, please see [2].

1.3.1. Communication Channels in Singularity

Singularity's channels mechanism implements interprocess communication based on message passing. It is a lot easier to properly implement interprocess communication using a message exchange protocol than by sharing memory between processes. The system requires explicit specification of the communication protocol, which enables static and dynamic verification of the communication. Static code analysis algorithms can analyze MSIL code and detect most errors associated with invalid use of channels, before the communicating processes are even started.

A communication channel consists of two channel endpoints, which are used to send and receive data through the channel. One endpoint is used to receive messages and the other one is used to send them. Each endpoint can be owned by at most one thread and only that thread can use the endpoint. Additionally, Singularity channels are bidirectional, so it suffices to have a single channel between two processes and enable communication between them in both directions.

Messages are sent through channels asynchronously. The send operation does not block the sending process and never requires additional memory. On the other hand, receiving messages from an endpoint can block the receiving process. Furthermore, the Sing# language provides special syntax, which allows to wait for messages which satisfy specified criteria.

From the programmers point of view, messages are just objects which contain data of strictly defined types. However, not every type of data can be sent through a communication channel. Singularity requires all data types, which are used in channel messages, to be simple types such as integers, bytes, boolean values, etc. Particularly references to objects cannot be passed through channels, because the referenced object exists only in the sending process and it does not make any sense in the other process. As mentioned earlier, this makes it impossible to share objects between processes in Singularity.

Some applications may require exchanging more complex data than simple types. To send complex objects through channels, the objects must be either serialized to some binary format, or represented as special structures called rep structs. Rep structs seem similar to regular structures in the .NET Framework, however they are handled completely differently by the operating system. First of all, rep structs are value types, which means that they are stored as values and not as references to data. Additionally, rep structs may contain only values of simple types or instances of other rep structs. This way they do not contain references, and thus they can be passed through communication channels. In contrast to standard structures in the .NET Framework, rep structs can use inheritance, however they can inherit only from rep structs.

Channel messages are stored in a special region of system memory, called the exchange heap. Objects on the exchange heap are stored in memory blocks called regions. They can be referenced by processes through allocations, which can be seen as type-safe pointers to regions in the exchange heap. Additionally, objects in the exchange heap can be handed off between processes. Each object is initially owned by the thread which created it and later the object can be passed over (with ownership) to another thread. Each object in the exchange heap is owned by at most one thread at the same time, so there is no memory sharing between processes. The described mechanism is used to send messages over channels. Such implementation allows passing objects between processes without the need to physically copy the data from one object space to the other.

The concept of storing messages on the exchange heap introduces a couple of problems. First of all, more than one process may access a single region of memory, which might allow memory (and state) sharing. Additionally, the exchange heap is not garbage collected and requires explicit memory management (a reference counting algorithm is used). This introduces the possibility of creating memory leaks. Both problems are solved by the Sing# compiler and static analysis tools, which must guarantee that processes do not share objects on the exchange heap and that they utilize exchange heap memory properly. The Sing# language introduces a special instruction, called expose. This instruction must be used to access data of channel messages and rep structs. There are many rules, which specify how the expose instruction may be legally used. For example a rep struct can be exposed only from a thread which owns the object. An interesting fact is that the expose instruction does not actually do anything (no code is generated by the compiler for the instruction). It is used by the compiler to statically check and guarantee that objects on the exchange heap are not shared between processes and that all memory is allocated and freed properly. If a program does not use the expose instruction properly or tries to access data on the exchange heap without using the expose instruction, a compile-time error is generated and the application does not compile successfully. More details about the expose instruction and its compile-time verification algorithms can be found in [2].

There is one exception from the rule that object references cannot be passed through channels. Namely channel endpoints may be sent through channels. The semantics of such an operation are defined as passing ownership of the channel endpoint to the receiving process. The sending process cannot use the endpoint after it has been sent to another process.

Furthermore, a channel endpoint can be sent to another process only if no send or receive operations were executed on the endpoint.

New communication channels can be created in one of two ways. They can be automatically created by the system when an application's processes are started (if the application is configured this way). The second way is by calling a system function, which creates a new channel and passes both endpoints to the calling process. In the second case, the process usually passes one of the endpoints to another process and uses the other endpoint to communicate with the other process.

1.3.2. Channel Contracts

As mentioned earlier, every channel has a strictly defined communication protocol. The definition of such a protocol is called a channel contract.

Channel contracts specify how communication can be carried out through the channel. First of all, a contract specifies the data types which can be sent through the channel. This specification consists of a set of message type definitions. Each message has a strongly-typed list of parameters (the data sent in through the channel) and each message is defined as incoming, outgoing or both. The contract is always defined from the exporting endpoint's view, so incoming messages will be received and outgoing messages will be sent. The definition of such a contract automatically specifies the contract of the other endpoint, in which incoming messages become outgoing and outgoing messages become incoming.

Additionally, each contract defines the order in which messages can be sent over the channel. This specification is in the form of a state machine definition. The contract specifies valid states in which the channel's endpoints can be, and it specifies the transition of states when a specified type of message is received or sent.

Listing 1 presents an example of a channel contract definition. The definition begins with the contract's name (`Addition`) in line `1`. Lines `3-4` define two messages. The first one, which is named `Parameters` is an incoming message and contains two integer values. The other message, called `Result`, is outgoing and contains a single integer value. Further, in lines `6-12`, two contract states are specified. The first state, which is the initial state of the contract, is called `Start`. In this state, the endpoint can receive a message of type `Parameters`, and in that case, the channel's state changes to `SendResult`. In the `SendResult` state, the endpoint can send a message of type `Result` and then the endpoint's state transitions back to the `Start` state. A process which uses this contract may, for example, implement an algorithm which adds two integer numbers and sends the result to the caller.

Channel contracts can extend other contracts in a way which resembles inheritance in object-oriented programming. Contracts can add new messages and new states to existing contracts and they can also override existing state transitions. There are some restrictions, for example a state can be overridden only if it always begins with sending a message (and not receiving one). Nevertheless, contract extensions enable defining generic protocols and using them polymorphically, like abstract classes in object-oriented programming. For example, if an application uses many protocols which have a common subset of message types and states, then a base contract can be defined and the application code can operate on the base contract, using only the common subset of states and messages. Later on in this thesis, examples of such generic contracts will be introduced in the description of Singularity's device driver model.

Contract definitions enable static and dynamic verification of communication over channels. First of all, by using static code verification, Singularity can check whether applications communicate according to their channel contracts, without executing any of

Listing 1: A simple contract definition

```
contract Addition
{
   in message Parameters(int a, int b);
   out message Result(int r);

   state Start : one {
      Parameters? -> SendResult;
   }

   state SendResult : one {
      Result! -> Start;
   }
}
```

the application's code. The system is able to detect whether applications exchange messages of valid types and whether the messages are sent in the order specified by the contract. Furthermore, it can be guaranteed that two processes which communicate with each other using a specified contract will never enter a deadlock state [2].

1.4. Singularity Kernel

This section briefly presents the architecture of Singularity's kernel and the main system components which are implemented as a part of the kernel.

1.4.1. Microkernel Implementation

Singularity's kernel is based on the microkernel concept. The idea means that the system's kernel is very small and contains only the most important system components. All other services, such as file systems, device drivers, etc. are implemented as external processes, which communicate with the system's kernel using interprocess communication mechanisms.

The microkernel architecture has many advantages. First of all, errors which occur in system components are well isolated from the rest of the system. In traditional operating systems, where most services are implemented in the system's kernel, an error in one of these system components leads to a system failure and requires the whole system to be restarted. In the case of microkernel systems, it suffices to restart only the single component, in which the error occurred. Additionally, in microkernel systems, it is usually possible to exchange system components dynamically, even without restarting the system (for example a system service can be stopped, and another implementation of the service can be started). In systems with services implemented as part of the kernel, such operation may even require recompiling the system's kernel.

Another advantage of microkernel systems, is that such systems must implement device drivers in separate processes and this leads to isolating driver errors from the rest of the system. Device drivers are one of the most unreliable types of software [7]. They are developed very rapidly and modern hardware is much more complicated than it used to be in the past. Due to the limited capabilities of testing complex device drivers, they usually contain many errors. Using modern hardware requires running new, often poorly tested drivers, because no other are available. This results in decreased stability of computer systems. Modern operating systems usually load device drivers into the kernel's code space, because such architecture makes it easy to build drivers, which work very efficiently. However, in such architecture, a device driver failure results in a failure of the entire operating system.

Although microkernel systems have many advantages, most popular and commonly used operating systems are based on large, monolithic kernels, which include most system services and components inside the kernel. The main reason of this fact is performance. Interprocess communication is expensive and in the case of heavily used system components (for example the input/output subsystem or device drivers), implementing them in separate processes introduces an overhead, which is unacceptable in commercial applications.

Singularity is designed as a microkernel system, although some system services, such as resource schedulers and memory management algorithms, are implemented inside the kernel. However, almost all device drivers are implemented as processes, whose failure does not threaten the system's stability.

The overhead related to the microkernel architecture in Singularity is a lot smaller then in the case of other operating systems. The reason for this fact is that Singularity's interprocess communication mechanisms and software isolated processes were designed particularly with the microkernel idea in mind. SIPs are lightweight processes, which can all run in privileged mode, so the context switch cost is much lower than in traditional operating systems. Furthermore, communication channels enable fast data exchange, without the need to copy data from one process to another. This is particularly important in the case of device drivers, which can generate a lot of data (for example the network adapter or mass storage device drivers). All this together makes Singularity's performance acceptable, at least for a research operating system.

1.4.2. Singularity Kernel Components

Resource Scheduling

Every reentrant operating system must efficiently manage available resources and share them among all executing processes. In Singularity, components which are responsible for resource scheduling are an important part of the system's kernel. Singularity implements many resource management algorithms. The algorithm which will be used by the system can be selected at kernel compile-time.

One of the most valuable resource in computer systems is the CPU time, which is required to execute any code. Singularity implements three algorithms to manage this resource. The first one is a standard round-robin algorithm, which periodically gives the CPU to processes which are ready for execution.

The second scheduling algorithm, called the Rialto algorithm [6], was developed as a part of another Microsoft Research project. It is a very complex scheduling algorithm, with features such as CPU time reservations for individual processes. The algorithm was designed for systems which support coexisting real-time and non-real-time processes. Singularity is not an operating system for hard real-time applications, however, the system has been designed to support soft real-time processes. Supporting such applications has recently become very important, due to the fact that more and more programs need to get system resources regularly to work properly. An example of such an application is a video player, which requires CPU time in regular intervals, to be able to decode video frames in time to continuously display a movie.

The third CPU scheduling algorithm is called the minimum latency round-robin scheduler. It is an implementation of the standard round-robin algorithm, designed specially for Singularity. The algorithm is optimized for systems with many threads, which often communicate with each other and sleep a lot. Processes in Singularity match this pattern very well. For more details on the algorithm, see [4], [5].

Input/Output Subsystem

The input/output subsystem is one of the most important components of Singularity. It is responsible for managing data flow between processes and the system's kernel. Particularly, the subsystem manages data transfers between device drivers, devices and the system's kernel. Additionally, the subsystem is responsible for detecting available devices and managing their drivers. The next chapter of this thesis contains a detailed description of Singularity's input/output subsystem.

Memory Management

Another important component of the Singularity system, is the memory management subsystem. This component includes the implementation of Singularity's garbage collection algorithms, communication channels and exchange heap management.

Both the processes executing in Singularity and the system's kernel, rely on automatic memory management. There are many garbage collection algorithms and it is difficult to select one, universal algorithm which would be efficient for all types of software. Due to the fact that processes in Singularity do not share memory, it is possible to implement garbage collection independently for each process. This enables applications to select an algorithm which best fits their needs and characteristics. Garbage collection algorithms are implemented as part of Singularity's kernel.

As mentioned previously, the exchange heap is a system memory region, which stores messages sent through communication channels. The exchange heap does not use garbage collection, yet it uses reference counting of regions. The operating system manages the number of allocations for each region and frees its memory when the number of allocations drops to zero.

Metadata Management

Singularity extensively uses various types of metadata to store information about the system, its components and configuration. Metadata is usually stored as XML documents. Some of the metadata is defined in MSIL code, using C#'s attributes mechanism [9]. However, this data is usually extracted during code compilation and stored in XML format.

First of all, Singularity introduces the application as a first-class operating system abstraction. In traditional operating systems, there is only the abstraction of a process. However, applications contain many processes, data, and metadata files and the lack of an application abstraction leads to many problems. For example, in most operating systems, the process of installing an application is usually limited to copying files, without the knowledge of the files relationships. This often leads to situations in which existing applications stop working after the installation of new software, because a file shared by many applications is changed by the installer. The application abstraction also enables flexible per-application permission management.

Applications are introduced to the system using manifests. Manifests are XML documents, which describe all the application's components and the relationships between them. Manifests declaratively describe the system's state with the application installed. During installation, this is the system's responsibility to determine which operations need to be executed, to bring the system to such a state. After installation, the application's manifest is stored in the system and is used to launch the application. Such a solution enables describing all the application's processes and communication channels between them. The system can then automatically set up all the processes when the application is launched.

Manifests for device drivers have a substantial meaning for Singularity. A device driver manifest declares what devices will be used by the driver, all necessary hardware resources (such as device ports, memory ranges, etc.), and supported channel contracts. Thanks to the manifest, Singularity is able to detect driver conflicts, even before the drivers are loaded. Furthermore, the system can monitor whether drivers use only resources which are declared in their manifests. This is usually impossible in traditional operating systems, in which device drivers are pieces of software which are loaded into the kernel and communicate directly with devices. Singularity's device driver model is described in detail in the next chapter of this thesis.

System Namespace

The system namespace in Singularity is a hierarchical set of entries, which are globally visible in the entire system. Every entry represents some sort of system abstraction (for example applications, device drivers, files, etc.). Processes can reference these abstractions using the system namespace entries.

Entries in Singularity's system namespace are stored in a binary format in contrast to many operating systems, which store namespace entries as text. Each entry consists of the entry's length and a sequence of bytes. Such representation enables convenient application of logical operators (such as and, or, xor, etc.) to namespace entries. Such operations are very helpful in some situations, for example in the case of network addresses.

Besides simple data entries, the system namespace is capable of storing other types of entries. Fist of all, since the namespace is organized hierarchically, collections of entires can be represented by entries which contain sub-entries. Such collection entries are called directories. Singularity's system namespace also support entries which reference other entries from the namespace. Such entries are called links. Furthermore, channel endpoints can be stored in the namespace.

The system namespace contains persistent entries which exist throughout the system's lifetime, as well as non-persistent entries which represent ephemeral abstractions, such as network connections or open files. Furthermore, the system namespace can be extended by various processes. For example bus drivers may add newly detected devices to the namespace.

Application Binary Interface

Most of Singularity's system services are exposed through channels. The system opens channels between processes and the kernel and the processes can then request services from the system by sending messages through these channels. The communication channels can be created by the system when the application is started. Additionally, some channels can be accessed through the system namespace.

However, some services in Singularity are exposed using a traditional binary interface – a set of system functions, which are implemented in Singularity as static methods. Singularity exposes system functions for channel management, process and thread creation, synchronization, etc. Particularly creating new channels and sending messages through channels are implemented using system functions.

Due to the fact, that the kernel has a separate object space, which cannot be shared with any process, all parameter values of system calls must be simple type values (and not object references).

System calls in Singularity can change the state of the calling process or one of its descendant processes only. Changing the state of another process in the system can be done only using channel communication. Such a solution enables processes to constraint the access of their child processes to the rest of the system by limiting their access to particular channels. Furthermore, the application binary interface between a process and the kernel cannot be intercepted or altered in any way without explicit approval of the process' author [4].

Chapter 2

Singularity I/O Subsystem

This chapter presents Singularity's input/output subsystem, its tasks and architecture. Additionally, this part of the thesis describes the device driver model which has been implemented in Singularity. Everything described in this chapter refers to the version of Singularity which was released by Microsoft as the Singularity Research Development Kit 2.0.

2.1. Introduction to Singularity's I/O Subsystem

The input/output subsystem is one of the most important components in the Singularity operating system. Without it, Singularity would be a closed black box without any input or output mechanisms. The design and architecture of the subsystem, including SIP-based device drivers and channel-based communication, are one of the many ideas which clearly distinguish Singularity from modern, commonly used operating systems.

2.1.1. I/O Subsystem Responsibilities

The main task of Singularity's input/output subsystem is to manage data flow between devices, device drivers and the system's kernel. Particularly device drivers which are implemented in separate SIPs, communicate with devices through the I/O subsystem, instead of accessing hardware directly, as in most operating systems. All data generated by devices (for example data read by a block device) or hardware state changes, are passed between drivers and the kernel through the I/O subsystem (usually via channels).

The input/output subsystem is also responsible for device management. During system startup, the subsystem must detect available devices and load drivers which are suitable for them. Device management during the system's lifetime (for instance loading drivers for dynamically attached hardware) should also be done by the I/O subsystem, however it is not yet implemented in Singularity.

2.1.2. I/O Subsystem Prerequisites

During the design of Singularity's I/O subsystem, several assumptions were made about the rest of the system. Some of these assumptions would not be acceptable in the case of commercially used operating systems, due to the lack of compatibility with existing applications or decreased performance. However, in the case of Singularity, which is only a research operating system, such assumptions can be made. This section summarizes the main ideas on which the I/O subsystem relies.

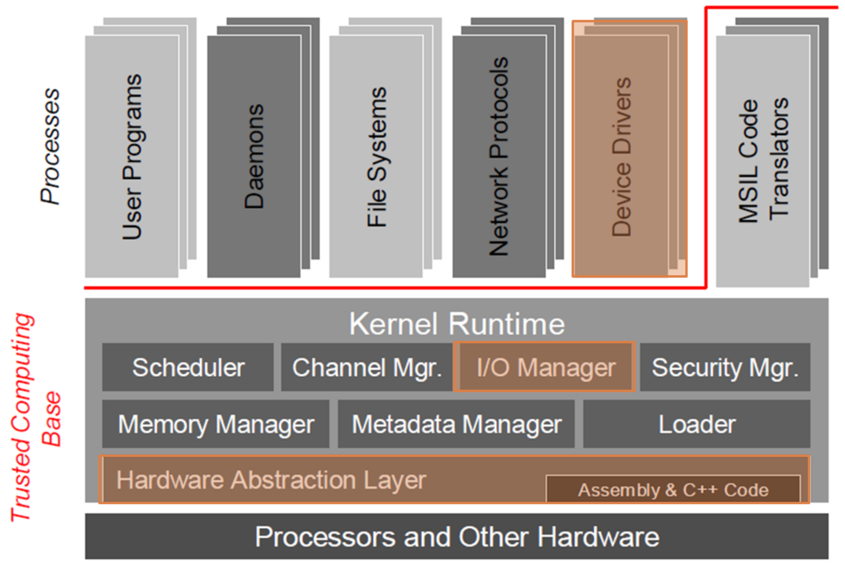

Figure 2.1: The architecture of the Singularity operating system (the I/O subsystem is marked as red) [3]

First of all, assuming that device drivers are implemented in separate processes, in order to provide performance acceptable even for a research operating system, Singularity must implement some mechanism of passing data between processes, which does not involve copying the data physically. Without such mechanism, implementing drivers in separate SIPs for devices such as network adapters or mass storage devices, would not make much sense because of an unacceptable overhead. That is why Singularity's communication channels are implemented in a way which allows handing off data (in the exchange heap) from one process to another and such operation does not allocate any memory, nor does it copy the transfered data. This mechanism is sometimes called zero-copy buffers.

The system namespace has great importance to the I/O subsystem, because many I/O-related entities, such as devices, device drivers, etc., are stored as namespace entries. The namespace must be extendible by external processes, particularly by device drivers which may need to add new entries to the namespace, For example, a network adapter driver may add entries for network connections and a hard disk driver may add entries for drive partitions. Additionally, the system namespace must be able to represent collections of entries, such as sets of devices connected to the same bus or sets of files which belong to the same directory. Such sets are represented in the namespace hierarchically (entries which contain sub-entries).

Another component of the system, which is very important to the I/O subsystem, is the memory management component. Singularity supports memory paging, however it does not support page swapping. The main reason for such an assumption, is to guarantee that memory will never be corrupted due to an I/O error, while reading or writing memory to a swap device. Such errors would be very dangerous for the system's stability, especially if the corrupted page contains executable code.

2.1.3. I/O Subsystem Architecture

Singularity's input/output subsystem consists of three parts – the Hardware Abstraction Layer, the I/O manager and the device drivers (see figure 2.1). The I/O manager uses methods implemented in the hardware abstraction layer to access hardware. Communication between the I/O manager and device drivers is mostly implemented using channels.

Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) is responsible for supporting low-level devices, such as the system timer, interrupt vector, Basic Input/Output System (BIOS), etc. Additionally, the HAL provides access to devices for higher layers of the I/O subsystem.

Hardware abstraction layers are usually implemented in operating systems to make these systems platform-independent. All platform-specific code is implemented in a low-level layer and the rest of the system relies on it. This way operating systems can be easily modified to work with different processors, platforms, etc. All the system developers must do to port a system to a different architecture, is to write a platform-specific implementation of the Hardware Abstraction Layer.

In Singularity, the Hardware Abstraction Layer is very strongly separated from the rest of the system, due to the fact that it must be partly implemented in unsafe programming languages. Besides C#, the HAL is implemented in C++ and x86 assembler. However, code in the mentioned low-level and unsafe programming languages amounts to only 5% of the system's kernel code [4]. The HAL is a part of Singularity's Trusted Computing Base (TCB). The TCB is a part of the system, which cannot be fully verified by static code verification mechanisms. The rest of the system is layered on the TCB. Of course the TCB should be as small as possible to minimize the probability of a system failure.

Singularity relies on the HAL as an interface to device memory, ports, interrupts, and other hardware support mechanisms. System components (particularly device drivers) cannot communicate directly with hardware, due to the fact that they must be implemented in managed code. Thus, they must communicate through the Hardware Abstraction Layer. This leads to a strong isolation of hardware support code from the rest of the system.

I/O Manager

The I/O manager is the central component of Singularity's input/output subsystem. The main task of the I/O manager is detecting and monitoring devices and loading drivers for them. Devices are detected by bus drivers, which are currently a part of the Hardware Abstraction Layer. Additionally, the hardware detection procedure is performed only during system startup. The I/O manager communicates with bus drivers to find out what hardware is available and then finds matching device drivers for each device. The driver loading procedure is described in detail later on in the thesis.

In case of a device driver failure, the I/O manager should restart the failed driver and restore the system's stability (for example by informing all other components about the newly loaded driver). Furthermore, none of the subsystem's components may fail due to the failure or unresponsiveness of another component. The subsystem's design assumes that such errors are handled gracefully.

Device Drivers

Device drivers are the third part of Singularity's I/O subsystem and this is the part of the subsystem which can be easily extended by adding new components to the system. Device drivers implement access to devices using precisely defined contracts, so that other parts of the system can utilize these devices. Each driver supports either a specific device or a group of devices.

In contrast to traditional operating systems, in Singularity device drivers are implemented in managed, type-safe programming languages and they do not communicate directly with hardware. Instead, they rely on the Hardware Abstraction Layer. A detailed description of Singularity's SIP-based driver model is presented later on in this chapter.

Although most Singularity device drivers are implemented as separate SIPs, some drivers, called kernel drivers, are implemented as part of the system's kernel. These drivers can be divided into two categories. The first one consists of drivers for low-level hardware, such as the system timer or interrupt vector. These drivers are implemented as part of the Hardware Abstraction Layer. The reason why these drivers are a part of Singularity's kernel, is that they support extremely low-level hardware and it would be very difficult and complicated to implement such drivers in high-level programming languages.

The second category of kernel drivers consists of bus device drivers, which are responsible for detecting the system's hardware configuration. These drivers are implemented as code which is called directly by the I/O manager. They are a part of the system's kernel to simplify their implementation.

2.2. SIP-based Device Driver Model

This section describes how device drivers are implemented in Singularity, how they are loaded and how they work in the system. The description in this section involves only SIP-based drivers, which are executed in separate processes and not kernel drivers.

2.2.1. Introduction

Most device drivers in Singularity are implemented using a new, experimental SIP-based model which is completely different from the device driver model used by traditional operating systems.

First of all, SIP-based device drivers are executed in separate processes, outside the system's kernel. Such process-based drivers have many advantages. Most of all, they are strictly separated from the kernel, so no error in the driver's code affects the kernel and the entire system. Furthermore, process-based drivers can be easily restarted in case of a failure. All these advantages result in a better system stability and dependability.

Despite the mentioned advantages of process-based device drivers, almost all popular, contemporary operating systems use a driver model in which device drivers ale loaded into the system's kernel. There are mainly two reasons for such solution. First of all, kernel drivers are usually a lot more efficient then process-based drivers, because of the lack of overhead caused by process context switching and hardware memory management. Such overhead is usually unacceptable in commercial operating systems in which efficiency is very important. In Singularity this is not a big issue, because SIPs can be executed in the same hardware protection domain as the kernel, and hardware memory management is not needed. The second reason why process-based drivers are not popular, is the fact that such a driver model requires efficient and convenient interprocess communication mechanisms. Shared memory or simple message queues offered by most operating systems is not enough. Singularity offers communication channels, which are easy to use and much less error prone then the mentioned mechanisms.

Listing 2: Part of the IoMemory class definition

```
 1 public sealed class IoMemory
 2 {
 3    private readonly PhysicalAddress  dataPhys;
 4    private readonly unsafe byte *    data;
 5    private readonly int              bytes;
 6    private readonly MemoryType       type;
 7    private bool                      readable;
 8    private bool                      writable;
 9    ...
10    public unsafe byte Read8(int byteOffset) { ... }
11    public unsafe ushort Read16(int byteOffset) { ... }
12    public unsafe uint Read32(int byteOffset) { ... }
13    public unsafe ulong Read64(int byteOffset) { ... }
14    ...
15    public unsafe void Write8(int byteOffset, byte value) { ... }
16    public unsafe void Write16(int byteOffset, ushort value) { ... }
17    public unsafe void Write32(int byteOffset, uint value) { ... }
18    public unsafe void Write64(int byteOffset, ulong value) { ... }
19    ...
20 }
```

2.2.2. Type-safe Hardware Communication

SIP-based device drivers are not only executed in separate processes, but are also represented to the system as type-safe MSIL code. This is quite an unusual approach, since device drivers require low-level programming instructions to communicate with hardware. SIP-based drivers cannot communicate directly with their devices, because MSIL does not provide such low-level instructions. That's why device drivers in Singularity must rely on the Hardware Abstraction Layer, which may contain unsafe code. The HAL provides type-safe abstractions of low-level primitives to the driver processes as objects which represent the device's memory, ports, etc.

The HAL implements several classes which represent hardware resources in a type-safe and object-oriented way. For example the `IoMemory` class represents a memory buffer. Some fragments of this class are presented in listing 2. The data stored in `IoMemory` objects are declared in lines `3-8`. The object simply encapsulates an unsafe memory pointer and stores some additional information such as the type of represented memory, and whether the memory is writable. The class implements several methods which can be used to access the encapsulated pointer from MSIL code in a type-safe way. For example the `Read16` method reads 16 bytes at the specified offset in the buffer.

There are many other objects which similarly represent hardware abstractions in a type-safe way. For example the `IoPort` class represents a hardware port, which can be used to communicate with devices and the `IoDma` class enables use of the Direct Memory Access mechanism from MSIL code.

2.2.3. Hardware Resource Allocation

The previous section of this chapter described how managed device drivers access hardware through system objects, which are implemented in the Hardware Abstraction Layer. It is yet still unclear how device drivers obtain or create these objects.

In most contemporary operating systems, in which drivers communicate directly with hardware, drivers acquire needed device resources themselves. This is possible due to the fact, that such device drivers are executed as part of the system's kernel with highest privileges possible. For example when a graphics adapter driver is loaded, it directly accesses a memory region on the graphics adapter which represents the data which will be displayed to the user. This solution has a major drawback – the operating system does not know which resources are used by which drivers, since drivers access resources directly, without informing the system about obtained resources. As a result, device drivers can access completely different hardware than they were installed for and the operating system cannot do anything about it. Furthermore, a driver may access a wrong device by error and bring the entire system down. The system's stability is also at risk in case of a resources allocation conflict, in which more than one driver accesses the same hardware resource (for example the same memory region).

To eliminate the mentioned drawback, the SIP-based driver model provides a completely different way of acquiring hardware resources by drivers. In order to access hardware resources, managed drivers require instances of system objects, which are implemented in the Hardware Abstraction Layer. Moreover, these objects can be created by kernel code only and cannot be passed between SIPs, since they are not rep structs. This leads to a model, in which device drivers must request resources from the operating system and the operating system grants such requests by creating system objects for the driver. This solution enables very accurate resource tracking – the system knows precisely what resources are used by which driver. Furthermore, if a driver requests a resource which is already used by another driver or is inconsistent with the drivers configuration (for example a graphics adapter driver requests resources of a block device), the operating system can deny such a request.

SIP-based device drivers are regular Singularity applications and as such they contain an application manifest. Manifests for device drivers contain additional data about hardware resources required by the driver. This section of the manifest is usually generated automatically during compilation, based on the resources declared by the driver.

In listing 3 is presented a complete declaration of hardware resources and communication channels for the S3Trio graphics adapter driver (communication channels will be discussed in the next section). The `S3TrioConfig` class, which stores all system objects for the driver's resources, contains several system object declarations. Each of the declared fields has an attribute[1] which specifies what resources will be represented by the object. For example the `frameBuffer` object (declared in line 6) represents a memory region in the graphics adapter device's memory, whose length is `0x400000` bytes and the default address of the region is `0xf8000000`. The `textBuffer` object (declared in line 9) is similar to the previous object, however, it represents a memory region at a fixed location in the graphics adapter's memory. The region's size is `0x8000` bytes and the region starts at address `0xb8000`. The `fontBuffer` is declared similarly. The ports which will be used for communication with the graphics adapter are declared in lines `14-21`.

When the driver is loaded and initialized, the operating systems verifies each of the specified resources. First of all, the system verifies whether the specified resources are not used by any other driver and whether they are consistent with the driver's manifest. If there are no conflicts, then the system objects (such as `IoMemoryRange` and `IoFixedPortRange`) are created by the kernel and references to the objects are stored in the proper fields. For example, the `IoMemoryRange` object which represents the text buffer is stored in the field declared in line `9`.

[1]The attributes mechanism is provided by the C# programming language to enable adding custom data to various language constructs, such as variables, classes, methods, etc. The information is included in the compiled MSIL code and is usually retrieved during code compilation or at run-time.

Listing 3: Resource and communication declarations for the S3Trio graphics adapter

```
[DriverCategory]
[Signature("pci/ven_5333&dev_8811&cc_0300")]
internal class S3TrioConfig: DriverCategoryDeclaration
{
   [IoMemoryRange(0, Default = 0xf8000000, Length = 0x400000)]
   internal readonly IoMemoryRange frameBuffer;

   [IoFixedMemoryRange(Base = 0xb8000, Length = 0x8000)]
   internal readonly IoMemoryRange textBuffer;

   [IoFixedMemoryRange(Base = 0xa0000, Length = 0x8000)]
   internal readonly IoMemoryRange fontBuffer;

   [IoFixedPortRange(Base = 0x03c0, Length = 0x20)]
   internal readonly IoPortRange control;

   [IoFixedPortRange(Base = 0x4ae8, Length = 0x02)]
   internal readonly IoPortRange advanced;

   [IoFixedPortRange(Base = 0x9ae8, Length = 0x02)]
   internal readonly IoPortRange gpstat;

   [ExtensionEndpoint]
   internal TRef<ExtensionContract.Exp:Start> ec;

   [ServiceEndpoint(typeof(VideoDeviceContract))]
   internal TRef<ServiceProviderContract.Exp:Start> video;

   [ServiceEndpoint(typeof(ConsoleDeviceContract))]
   internal TRef<ServiceProviderContract.Exp:Start> console;

   internal int DriverMain(string instance) {
      return S3Control.DriverMain(this);
   }
}
```

To sum up, the only way SIP-based device drivers can access hardware resources, is by specifying the resources in their application manifest. This in turn provides the system with precise information about what resources are used by which drivers. The system is responsible for providing the driver with system objects encapsulating the specified hardware resources when the driver is initialized. Resources can be easily declared in driver code using C# attributes and the driver's manifest can be generated automatically when the driver is compiled to MSIL code.

2.2.4. Channel Communication

All SIP-based device drivers use channels to communicate with their environment. The I/O manager controls drivers through a special channel and requests for the driver are delivered through another channel. This section describes what contracts are used by managed device drivers and how incoming messages are serviced by them.

Listing 4: ServiceContract definition

```
contract ServiceContract
{
   out message ContractNotSupported();

   state Start : one {
      ContractNotSupported! -> DoneState;
   }

   state DoneState : one { }
}
```

Singularity's SIP-based driver model heavily relies on the contract extension mechanism. Device drivers define their capabilities by contracts which extend other contracts and the base contracts define capabilities which are common for several device drivers. This enables the system to treat device drivers generically if there is no need to use driver-specific capabilities. Such a mechanism also simplifies implementing special (for example hardware-specific) driver features. For example a graphics device driver can implement some hardware-specific operations, by defining them in a contract which extends Singularity's standard video driver contract. This way the system and all applications can use the driver as a regular video driver through the standard contract and the special operations can be accessed through the new, driver-specific contract.

The most important base contracts will now be introduced. The `ServiceContract` (presented in listing 4) is the base contract for all contracts used to define driver capabilities. It can be seen as a base, abstract contract, similar to the Object class in object-oriented programming languages. The contract does not support any operations, all it does is sending `ContractNotSupported` messages. Contracts which extend the `ServiceContract` must override the start state and provide new states which send and receive meaningful messages. The purpose of the `ServiceContract` is to represent a generic service.

The `ExtensionContract` (presented in listing 5) extends the `ServiceContract` and adds initialization and shutdown support. The contract sends a `Success` message after successful initialization. Subsequently it waits for a `Shutdown` message to perform deinitialization. Finally it sends a `AckShutdown` or `NakShutdown` depending whether the deinitialization was completed successfully.

The next important contract is the `ServiceProviderContract` (presented in listing 6). It is used by applications which provide services that can be accessed by several clients and it is also used by device drivers to accept requests from their environment. Applications which expose services using the `ServiceProviderContract`, register their endpoints in the system namespace. An application which wishes to use the service must send a `Connect` message and pass a channel endpoint which extends the `ServiceContract`. If the service provider supports the specified channel contract, then it will send a `AckConnect` message in reply. If the specified contract is not supported, then a `NackConnect` will be sent (with the channel endpoint specified in the `Connect` message). Subsequently, the service provider will communicate with the application using the specified endpoint. Obviously, the service provider must support simultaneous communication with many applications, through several channels.

Listing 7 presents a fragment of Singularity's video driver code. This code will be used to present how SIP-based device drivers process requests through channels. Note that some

Listing 5: ExtensionContract definition

```
contract ExtensionContract : ServiceContract
{
   in message Shutdown();
   out message AckShutdown();
   out message NakShutdown();
   out message Success();

   override state Start : one {
      Success! -> Ready;
   }

   state Ready : one {
      Shutdown? -> (AckShutdown! or NakShutdown!) -> Ready;
   }
}
```

Listing 6: ServiceProviderContract definition.

```
contract ServiceProviderContract
{
   in message Connect(ServiceContract.Exp:Start! exp);
   out message AckConnect();
   out message NackConnect(ServiceContract.Exp:Start exp);

   state Start : Connect? -> Ack;

   state Ack : one {
      AckConnect! -> Start;
      NackConnect! -> Start;
   }
}
```

less important code fragments have been omitted from the listing to increase readability.

After initialization, in line 3, the driver sends a `Success` message through the `ec` channel, which implements `ExtensionContract`. It is the same channel which is declared among the driver's resources in listing 3. This informs the I/O manager, that the driver has been successfully initialized.

Next, in lines 5-6 is the declaration of the `vs` collection, which is a set of channel endpoints, which implement the contract `VideoDeviceContract`, Singularity's standard video driver contract. Furthermore, endpoints stored in the set must be in the `Ready` state. The collection will be used to store endpoints from clients which connect with the driver through the `ec` channel.

Lines 10-39 contain a `for` loop which is iterated as long as the boolean `run` variable is set to true. The loop will process all requests received by the driver's channels. Inside the loop is used a `switch receive` construct, which receives messages from one or more channels. Each type of message is handled in a separate `case` clause.

The `Shutdown` message of the `ExtensionContract` channel (`ec`) is handled in lines 13-16. A `AckShutdown` message is replied over the `ec` channel and the `run` variable is set to `false` to stop processing messages in the outer `for` loop.

Lines 18-29 handle the `Connect` message of the `ServiceProviderContract`, which is sent by applications that wish to use the driver. The `Connect` message contains a channel endpoint which implements a contract which extends `ServiceContract`. The driver supports only the Singularity's standard video driver contract, so the `as` operator is used to dynamically check whether the endpoint implements `VideoDeviceContract`. If the video contract is implemented by the endpoint, then the `newClient` variable will be a valid reference to the video contract channel, otherwise, the variable will be `null`. If the variable is `null`, then a `NackConnect` is sent to reject the incoming connection (in line 27). Otherwise, the new client endpoint is added to the `vs` collection and a `AckConnect` message is sent over the `ServiceProviderContract` channel.

The rest of the `case` clauses of the `switch receive` construct handle messages of the `VideoDeviceContract`. For example the contract's `Plot` message, which draws a point on the screen, is handled in lines 20-24. Note that the `Plot` messages are received from one of the endpoints in the `vs` collection, using Sing#'s `in` operator. The `switch receive` construct is implemented in a way that guarantees fairness. In other words, none of the client endpoints in the `vs` collection will have to wait infinitely long to be processed.

After the `for` loop is terminated by a `Shudown` message, the `ec` and `ve` channel endpoints are closed using the `delete` operator. Subsequently, the `vs` collection is disposed, which will lead to closing all client endpoints.

2.3. System Namespace Layout

This section describes how device and driver data is organized in Singularity's system namespace. Although the namespace contains information related to many aspects of the system, such as security, system services and diagnostic information, only the parts of the namespace related to devices and drivers are presented here.

One of the most important parts of the system namespace, related to the I/O subsystem, is the `/dev` directory. It contains symbolic links to channel endpoints for each loaded driver. These entries are used by the rest of the system to connect and communicate with the drivers. That's why these entries are called public endpoint names. Note that these entries are only links to the entries which represent the real endpoints (which are stored in the subdirectories of the `/hardware` tree).

The layout of a sample `/dev` directory is presented in listing 8. There are four public names in the namespace. The `disk0@` and `disk1@` entries point to the `DiskDrive` driver. Note that the links point to endpoints of different instances of the driver. Additionally, there is a link for the keyboard driver (`keyboard@`) and the network adapter device driver (`nic0@`).

Most of the I/O-related namespace entries are stored in the `/hardware` directory, which contains three important subdirectories. The `/hardware/devices` directory stores information about available devices. Each device is represented in the tree by a subdirectory, which contains entries specifying the device's driver and to which bus the device is connected. The `/hardware/drivers` directory stores information about devices drivers which have been registered in the system. Each driver instance contains a separate subdirectory in the tree. The `/hardware/registrations` directory contains entries which map device signatures to the drivers which support the devices specified by the signatures. The layout of a sample `/hardware` directory is presented in listing 9. Due to the enormous size of a real `/hardware` directory, only the most important subdirectories and entries have been presented.

The `/hardware` tree is built in two phases. The first phase consists of building subdirectories based on the system configuration and device information provided by

Listing 7: Channel processing in S3Trio driver

```
...

ExtensionContract.Exp! ec = resources.ec.Acquire();
ServiceProviderContract.Exp! ve = resources.video.Acquire();
ServiceProviderContract.Exp! te = resources.console.Acquire();

ec.SendSuccess();

ESet<VideoDeviceContract.Exp:Ready> vs
   = new ESet<VideoDeviceContract.Exp:Ready>();

...

for(bool run = true; run;) {
   switch receive {

      case ec.Shutdown():
         ec.SendAckShutdown();
         run = false;
         break;

      case ve.Connect(candidate):
         VideoDeviceContract.Exp newClient
            = candidate as VideoDeviceContract.Exp;
         if(newClient != null) {
            newClient.SendSuccess();
            vs.Add(newClient);
            ve.SendAckConnect();
         }
         else {
            ve.SendNackConnect(candidate);
         }
         break;

      case ep.Plot(x, y, color32) in vs:
         device.Plot(x, y, new RGB(color32));
         ep.SendAckPlot();
         vs.Add(ep);
         break;

      ...
   }
}

delete ec;
delete ve;
vs.Dispose();
```

Listing 8: Sample system namespace layout in Singularity – dev directory

```
+ <dir> dev
| + <chan> conout
| + <link> disk0@ -> /hardware/drivers/DiskDrive/instance0/endpoint1
| + <link> disk1@ -> /hardware/drivers/DiskDrive/instance1/endpoint1
| + <link> keyboard@ -> /hardware/drivers/LegacyKeyboard/.../endpoint1
| + <link> nic0@ -> /hardware/drivers/Tulip/instance0/endpoint1
```

bus drivers. Firstly, the `/hardware/drivers` directory is created based on the system configuration. A subdirectory in the tree is created for each installed driver. Then for each driver, a symbolic link named `image@` is created. It points to the directory which contains the driver's manifest and binary files, which are usually stored in the `/init` directory. Next, bus drivers are used to detect all devices present in the system and a subdirectory in the `/hardware/devices` and `/hardware/registrations` trees are created for each device. Each device has a signature, which specifies its type and the bus to which the device is connected. The names of the subdirectories usually depend on the device's signature. For example the keyboard, which has the signature `/pnp/PNP0303`, is represented in the namespace by the directory `/hardware/devices/pnp/PNP0303/instance0`.

The second phase of building the `/hardware` tree occurs when device drivers are activated. For each driver which has to be started, the system finds the driver's root directory (such as `/hardware/drivers/LegacyKeyboard`) and the root directory of the device which will be supported by the driver (such as `/hardware/devices/pnp/PNP0303/instance0`). First of all, a directory is created which will represent the new instance of the driver. For example, for the keyboard driver, such a directory might be `/hardware/drivers/DiskDrive/instance0`. Such naming enables running many instances of the same driver, which might be required if there are many instances of the same device type connected (for example many hard drives of the same model). Next, the driver's SIP is started based on the application manifest from the driver's directory, which is determined by the `image@` link in the driver's root directory. At this point the driver instance must be associated with its device instance. This is done by creating two symbolic links. The first one, called `device@`, is created in the driver's instance directory and points to the device's instance directory in the `/hardware/devices` tree. The second one, called `driver@`, is created in the device's instance directory and points to the driver's instance directory. Further, all channel endpoints of the driver are set up, which completes the driver's activation procedure. Now that all channel endpoints have been created, the devices' public names can be created in the `/dev` directory.

2.4. Driver Loading Procedure

The driver loading procedure, which occurs during system startup, consists of two phases – registration and activation. This part of the chapter describes how they work.

2.4.1. Driver Registration

The purpose of the registration phase is to find out what drivers are available in the system and what devices they can support.

First, all drivers compatible with the current hardware architecture are found by looking through the system-wide manifest. For each found and compatible driver, an entry is created in the system namespace, in the `/hardware/drivers` node. Next, the system discovers what

Listing 9: Sample system namespace layout in Singularity – hardware directory

```
+ <dir> hardware
| + <dir> devices
| | | ...
| | + <dir> ata
| | | + <dir> controller
| | | | + <dir> instance0
| | | | | + <link> driver@ -> /hardware/drivers/DiskDrive/instance0
| | | | | + <link> location@ -> /hardware/locations/acpi0/.../controller0
| | | | + <dir> instance1
| | | | | + <link> driver@ -> /hardware/drivers/DiskDrive/instance1
| | | | | + <link> instance@ -> /hardware/locations/acpi0/.../controller1
| | | ...
| | ...
| + <dir> drivers
| | + <dir> DiskDrive
| | | + <link> image@ -> /init/DiskDrive
| | | + <dir> instance0
| | | | + <link> device@ -> /hardware/devices/ata/controller/instance0
| | | + <dir> instance1
| | | | + <link> device@ -> /hardware/devices/ata/controller/instance1
| | | | + <chan> endpoint1
| | + <dir> kernel
| | + <dir> LegacyKeyboard
| | | + <link> image@ -> /init/LegacyKeyboard
| | | + <dir> instance0
| | | | + <link> device@ -> /hardware/devices/pnp/PNP303/instance0
| | | | + <chan> endpoint1
| | ...
| + <dir> registrations
| | + <dir> ata
| | | + <dir> controller
| | | | + <link> driver@ -> /hardware/drivers/DiskDrive
| | + <dir> pci
| | | ...
| | + <dir> pnp
| | | ...
| | + <dir> root
| | | + <dir> acpi0
| | | | + <link> driver@ -> /hardware/drivers/kernel
| | | ...
| | ...
| ...
```

devices can be supported by the driver, by finding all device signatures declared by the driver in its manifest. The discovered information is registered in a data structure maintained by the I/O manager.

When the mentioned procedure is completed, all SIP-based device drivers are registered in the system. The next step is to register internal, kernel drivers. This is done statically in kernel code, by registering signatures of kernel-supported devices and mapping them to proper kernel drivers.

2.4.2. Driver Activation

The next phase involves finding what devices are present in the system and loading proper drivers for them. In this phase also device instances are resolved with their drivers.

The driver activation phase is executed iterationally. In each iteration a driver which can be activated is found, it is associated with a device instance, initialized and loaded. A driver can be activated if all its dependencies are satisfied and there is a device in the system with a matching signature and without a driver.

A driver may specify more than one signature and it does not have to match the device signature exactly. This enables drivers to support generic groups of hardware. For example the IDE bus driver specifies the signature `pci/cc_0101`, which partially matches the device signature `pci/ven_8086&dev_7111&cc_0101&subsys_00000000&rev_01`. During driver activation, the I/O manager tries to find a driver with an exactly matching signature and if such a driver does not exist, then it tries to find a driver with the best partial signature match.

If a driver which was just activated supports a bus device, then it is used to enumerate the bus and discover new devices in the system. The following iterations associate the new devices with compatible drivers (if drivers are available for them) and activated. The first bus device, called the ACPI PNP bus, is registered by the kernel in an early system startup phase. This bus device enumerates the Basic Input/Output System (BIOS) to find all basic hardware. Later on, during the driver activation phase, additional buses, such as the PCI bus, are discovered.

The activation phase is completed when there are no more drivers to activate.

Chapter 3

SIP-based Bus Driver Model Implementation

The previous chapters presented the major ideas behind Singularity and the I/O subsystem. This part of the thesis describes the main effort of this thesis – a new bus driver model for Singularity, which allows to execute bus drivers in separate software isolated processes. This chapter describes how the original I/O system was extended to support the new driver model and presents details of the model's implementation. Finally, the last section of this chapter presents tests which were performed on the that implementation.

3.1. Singularity Bus Drivers Overview

The Singularity RDK 2.0 contains six bus drivers. The following sections briefly describe the purpose of each of those drivers and summarize which drivers were implemented using the SIP-based bus driver model.

3.1.1. Kernel Bus Drivers

The Pseudo bus driver is not actually a real device driver. It allows to register pseudo-devices which are typically layers in the I/O stack which may be associated with physical devices. However, at the time of the RDK 2.0 release, the Pseudo driver was used only for testing purposes.

The ACPI and PNP drivers support the Basic Input/Output System to discover basic low-level hardware. Only one of these drivers is used depending on the actual hardware platform.

Furthermore, there is a driver to support the PCI bus and an additional driver to support Low-Pin Count PCI devices. The latter is used to reroute integrated peripheral interrupts.

Finally, there is a driver to support the IDE/ATA bus controllers. Actually there are three variations of the driver, one of which is used depending on the actual hardware.

3.1.2. SIP-based Bus Drivers

Two of the mentioned drivers were successfully implemented using the new SIP-based bus driver model – namely the PCI and IDE drivers.

The ACPI/PNP drivers are used at a very early stage in the system's startup process, in which the I/O system is not yet initialized completely and therefore the driver cannot be executed using the new model. The case of the Pseudo bus driver is analogous.

The Low-Pin Count PCI driver was not re-implemented due to the lack of hardware on which the driver could be tested.

From the three variations of IDE drivers, the most common one (the one which is used on the virtual platform on which Singularity is usually used) was successfully implemented using the new driver model. The rest of the IDE drivers could also be easily implemented in the new model. However, similarly to the Low-Pin Count PCI driver, the implementations were not done due to the lack of hardware to test them.

3.2. SIP-based Bus Driver Model

This part of the thesis describes the new SIP-based bus driver model. The main objective of the new model is to improve system stability by protecting it from bus driver crashes.

3.2.1. Problem Overview

The basic difficulty in implementing bus drivers outside the kernel, is the fact that these drivers, in the kernel implementation, create HAL resource objects for discovered hardware. This obviously cannot be done outside the kernel. Therefore to implement bus drivers in SIPs, the discovery process must be divided between the driver and kernel, so that the driver reports the new hardware configuration and the kernel creates HAL objects which represent the hardware's resources.

Another problem is that some bus drivers rely on information about the devices in the system and their resources. This information is easily available inside the kernel, however it is limited in SIPs due to Singularity's driver model which is designed to minimize the information available to particular drivers. Therefore the question arises how this required information can be conveyed to bus drivers.

3.2.2. Model Overview

The purpose of a SIP-based bus driver is to find out the hardware configuration of the devices connected to its bus and report this information to a kernel component called the bus manager.

Similarly to other SIP-based drivers, bus drivers communicate with bus devices using resource objects implemented in the Hardware Abstraction Layer. These objects are allocated by the kernel at driver startup, accordingly to the hardware resources specified in the bus driver's manifest.

Every SIP-based bus driver must use at least two channels to communicate with the kernel. The first channel implements a `ServiceProviderContract` to communicate between the driver and the I/O manager. However, this channel is not used to exchange hardware configuration information between the driver and the kernel. The `ServiceProviderContract` is meant to implement services which can be used by applications running in Singularity, and bus drivers should be used only by the kernel. Additionally the service provider channel is used to unify the startup and communication across all SIP-based drivers in Singularity (the SIP-based driver model requires that all drivers have such a channel).

Information about the hardware configuration discovered by the bus driver is sent to the bus manager using a dedicated channel which implements the `BusControlContract` (the contract is discussed in detail in the next subsection). This channel is called the bus control channel.

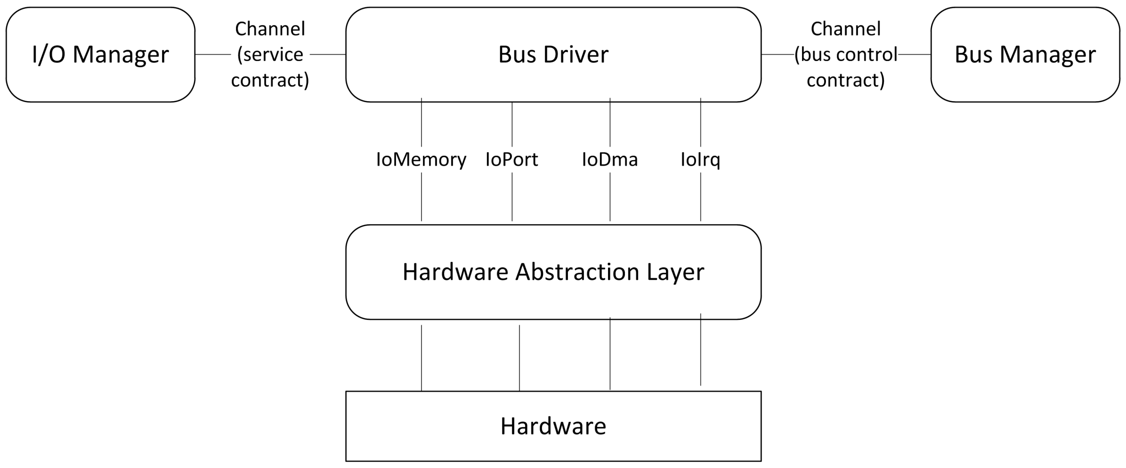
I/O Manager
Channel (service contract)
Bus Driver
Channel (bus control contract)
Bus Manager
IoMemory
IoPort
IoDma
IoIrq
Hardware Abstraction Layer
Hardware

Figure 3.1: SIP-based Bus Driver Communication

After the bus driver is started, it communicates with the bus device to find out what devices are connected to the bus. Then it sends this information over the bus control channel to the bus manager, which registers the devices and loads drivers for the new devices. After this initial bus enumeration, the bus driver may terminate or it may continue to monitor the bus and notify the bus manager when new devices are connected to the bus. This makes sense only for bus devices which support hot-pluggable devices (for instance the USB bus). Unfortunately such buses are not yet implemented in Singularity, so all of the implemented bus drivers exit after the initial enumeration, which occurs during system startup.

The communication model of SIP-based bus drivers is presented in figure 3.1.

3.2.3. Bus Device Driver Contract

Now the contract of bus control channels will be discussed. The definition of the contract is presented in listing 10.

Each bus driver has two phases of execution. The first phase is when the driver performs an initial enumeration of the devices connected to the bus. The second phase is when the driver monitors the bus for newly connected devices.

After a bus driver is successfully initialized, it sends a `Success` message to the bus manager and begins the first phase of execution. For each discovered device, the driver sends a `DeviceFound` message which contains information about the device's configuration in XML format (the format of this XML document is discussed in the next subsection). The bus manager acknowledges each device by sending a `DeviceFoundAck` message or rejects the device by sending a `DeviceFoundNack` message. Devices may be rejected if they conflict with other devices in the system or due to invalid data specified by the bus driver. After all devices have been reported to the bus manager, the driver sends an `EnumerationComplete` message and waits until the bus manager acknowledges it. This completes the first phase of execution and begins the second one.

In the second phase, the driver monitors its bus device and sends a `DeviceFound` message to the bus manager when a new device is discovered. Since not every bus device supports dynamically attached hardware, bus drivers which do not support monitoring the bus device may notify the kernel that they are no longer needed, by sending a `Finished` message to the bus manager. The kernel will acknowledge this message by replying with a `FinishedAck` or

Listing 10: The bus control channel contract

```
public contract BusControlContract
{
   out message Success();

   out message DeviceFound(char[]! in ExHeap deviceInfo);
   in message DeviceFoundAck();
   in message DeviceFoundNack();

   out message EnumerationComplete();
   in message EnumerationCompleteAck();

   in message Shutdown();
   out message ShutdownAck();
   out message ShutdownNack();

   out message Finished();
   in message FinishedAck();
   in message FinishedNack();

   state Start: one {
      Success! -> Enumerating;
   }

   state Enumerating: one {
      DeviceFound! -> (DeviceFoundAck? or DeviceFoundNack?)
                   -> Enumerating;
      EnumerationComplete! -> EnumerationCompleteAck?
                           -> Enumerated;
   }

   state Enumerated: one {
      DeviceFound! -> (DeviceFoundAck? or DeviceFoundNack?)
                   -> Enumerated;
      Shutdown? -> (ShutdownAck! or ShutdownNack!) -> Done;
      Finished! -> (FinishedAck? or FinishedNack?) -> Done;
   }

   state Done : one { }
}
```

`FinishedNack` message. In the first case, the driver SIP may exit, otherwise, it must stay running. The kernel should never send a `FinishedNack` message. However, the protocol leaves such an option, should it be required in the future. An active bus driver must be ready to receive a `Shutdown` message, which is a request from the kernel to shut down the driver. When the driver is deinitialized successfully it must send a `ShutdownAck` message and exit. In case of an error (for example if it is unable to shutdown the bus device or release used resources), the driver should send a `ShutdownNack` message. The `Shutdown` messages are usually sent to bus drivers when the system is shut down.

3.2.4. Hardware Information Exchange Format

The DeviceFound message of the BusControlContract contains an array of characters which specify an XML document describing the discovered device.

Each DeviceFound message describes an object of the IoConfig class, which is implemented in Singularity's hardware abstraction layer. Such an object encapsulates the device's signature, location and its fixed and dynamic resources, represented as HAL resource objects. The kernel versions of bus drivers instantiate these classes directly. In the SIP-based bus driver model, these objects are instantiated by the bus manager, accordingly to the device information sent by the bus driver.

Most devices are represented using the generic PnpConfig class, which just encapsulates many ranges of resources. This object encapsulates a range of HAL resource objects and device identifiers. Listing 11 presents a sample document which describes a PnpConfig object representing a disk controller device connected to the IDE bus.

The root node of the document describes a single IoConfig object. In the case of the example document in the listing this is an object of type PnpConfig (from the Microsoft.Singularity.Io namespace). The type of the object is declared explicitly as an attribute of the root element. Stating the type explicitly makes the bus manager implementation resistant to changes in the Hardware Abstraction Layer. For example, if new IoConfig classes are implemented to support new hardware, then new bus drivers can use the new objects without any changes in the bus driver support code. Additionally, the attributes of the IoConfig element specify the name of the bus, to which the device is connected and the device's signature and location. The Identifiers subnode specifies a list of identifiers associated with the device. In the listing there is only one identifier specified – /ata/controller. Further, there is a list of XML elements which describe HAL resource objects used by the device. In the case of the IDE controller in the listing, there is a single interrupt and three ranges of hardware ports. The types of the HAL resource objects are declared explicitly for extensibility reasons (for example, in the future Singularity may represent hardware ports using many classes which differ in behavior).

Devices connected to the PCI bus are represented by specialized PciConfig objects. These objects specify only one identifier and they encapsulate a PCI port which consists of an address port and a data port. The Hardware Abstraction Layer supports three types of PCI objects – the PciDeviceConfig, PciBridgeConfig and PciCardbusConfig. An example of an XML description of a PciDeviceConfig object is presented in listing 12.

3.3. SIP-based Bus Driver Model Implementation

This section of the thesis presents how the SIP-based bus driver model was implemented in the Singularity RDK 2.0 version.

3.3.1. The Bus Manager

Most of the newly written code is part of the new bus manager component. Actually, this code could be a part of the I/O manager, however to keep the architecture of Singularity's I/O system clean, the code was isolated into a separate component, which is used by the I/O manager. The bus manager is primarily responsible for two tasks – starting SIP-based bus drivers and registering in the system devices which were discovered by bus drivers.

Bus drivers are started by the BusManager.StartBusDriver method. The bus manager is designed to allow starting only one bus driver at a time. This means that bus drivers cannot

Listing 11: An XML document which describes a generic PNP device

```
<IoConfig Bus="ide"
          Signature="/controller0"
          Location="/pci/bus0000/dev0007/func0001/controller0"
          Type="Microsoft.Singularity.Io.PnpConfig">
   <Identifiers>
      <Identifier Id="/ata/controller" />
   </Identifiers>
   <Ranges>
      <Irq Type="Microsoft.Singularity.Io.IoIrqRange"
           Line="14" Size="1" />
      <Ports1 Type="Microsoft.Singularity.Io.IoPortRange
              Port="496" Size="8"
              Readable="true" Writable="true" />
      <Ports2 Type="Microsoft.Singularity.Io.IoPortRange
              Port="1012" Size="4"
              Readable="true" Writable="true" />
      <Ports3 Type="Microsoft.Singularity.Io.IoPortRange
              Port="65440" Size="8"
              Readable="true" Writable="true" />
   </Ranges>
</IoConfig>
```

Listing 12: An XML document which describes a device connected to the PCI bus

```
<IoConfig Bus="pci" Identifier="57"
          Signature="/bus0000/dev0007/func0001"
          Location="/pci/bus0000/dev0007/func0001"
          Type="Microsoft.Singularity.Io.PciDeviceConfig">
   <AddressPort Type="Microsoft.Singularity.Io.IoPort"
                Port="3320" Size="4"
                Readable="False" Writable="True" />
   <DataPort Type="Microsoft.Singularity.Io.IoPort"
             Port="3324" Size="4"
             Readable="True" Writable="True" />
</IoConfig>
```

be started simultaneously from multiple threads (in the current implementation they are started only at system startup). The reason for such a design is to simplify synchronization to various kernel structures and resources.

The `StartBusDriver` method uses Singularity's standard mechanism for starting SIP-based drivers. Before the driver's process is started, the bus manager must create all channels and HAL resource objects required by the driver. Resource objects are allocated accordingly to the resources defined in the driver's manifest. As far as channels are concerned, always two endpoints are binded to the driver SIP at startup – the service provider and the bus control channel endpoint. Obviously, the driver may create new channels explicitly after startup. Additionally, after the bus driver's process is started, the bus manager updates the system namespace (for example the public names are created it the `/dev` directory).

Next, the bus manager performs the initial device enumeration accordingly to the bus control channel contract, which was described earlier. Each newly discovered device is registered in the I/O system, so that it can be activated in future iterations of the driver activation process. In the current implementation, the bus manager assumes that bus enumeration occurs only during driver activation at system startup. To support hot-swap attached devices, the bus manager would have to restart the driver activation procedure after registering new devices. However, this is not yet supported by Singularity's I/O system.

When the bus driver finishes enumerating devices, the manager stores the driver's bus control channel endpoint on a list, which is handled by a special kernel thread, called the bus handler thread. This thread monitors all endpoints on the list (by using a `switch receive` statement, which was introduced in chapter 1 of the thesis) for incoming messages. When a message is received from the bus driver, it is handled by the bus handler thread. According to the bus control contract, the driver may send one of the two messages. It can either send a `DeviceFound` message to register a newly discovered device or a `Finished` message to terminate its execution. Since registering new devices after system startup is not supported, the bus manager always sends a `DeviceFoundNack` reply to a `DeviceFound` message. In case of a `Finished` message, it is acknowledged by the bus manager and the driver's endpoint is removed from the endpoints list (since no more messages may be received from the endpoint).

When the system is shutting down, the bus manager terminates the bus handler thread and sends a `Shutdown` message to all endpoints from the list which is monitored by the thread. This causes all active bus drivers to release any used resources and terminate their processes.

3.3.2. The Bus Driver Helper Library

The communication mechanism described in the previous section, requires relatively many lines of code to implement bus drivers. Furthermore, much of this code is similar and duplicated in each bus driver. That is why the new bus driver model implementation introduced a kernel library, called the Bus Driver Helper Library, which greatly simplifies writing SIP-based bus drivers. Currently, the library supports only bus drivers which enumerate their bus devices and finish their execution. However, it should be possible to extend the library to support drivers which monitor bus devices after startup, when Singularity's I/O system fully supports such drivers. This subsection introduces the Bus Driver Helper Library and describes how bus device drivers can be implemented using the library.

Bus drivers implemented using the Bus Driver Helper Library are comprised of at least three classes. For two of them, the library provides super-classes, which implement most of the functionality associated with channel communication with the bus manager. Driver implementors only need to create subclasses and override some methods to implement the logic of device enumeration.

The DriverCategoryDeclaration class

The first class, which has to be implemented by the driver is the `DriverCategoryDeclaration` class. This class is actually a part of Singularity's generic driver model and it is implemented by all Singularity drivers. It encapsulates the channels and HAL resource objects used by the driver and it specifies the signatures of devices which the driver supports. Furthermore, by convention, this class also implements the entry point of the driver process, called `DriverMain`. Similarly to other Singularity drivers, this class is used to build the application's manifest which declares the driver's resources and service channels.

The BusDriver class

The second required class is a subclass of the `BusDriver` class, which is implemented in the Bus Driver Helper Library. This class implements the entire channel communication with the bus manager.

Usually, drivers need to override only one method of this class – the `CreateEnumerator` method, which returns a `BusEnumerator` object for the driver (which is discussed later in this chapter). The method's single argument is the `DriverCategoryDeclaration` object which encapsulates the device's resources. Typical implementations pass this object to the `BusEnumerator`, which then uses the HAL resource objects to perform device enumeration.

Drivers can override two additional methods if required. The `Initialize` method is called after the `BusEnumerator` object has been created (by the `CreateEnumerator` method). It has three parameters – `DriverCategoryDeclaration` object, the bus control channel endpoint and the `BusEnumerator` object. This method may perform required initialization or validation. It may also stop the driver from performing bus enumeration (for instance in case of unsatisfied hardware state conditions) by returning `false` as its result. Otherwise, the method must return `true`.

The `Finalize` method is called after device enumeration. Drivers can override it to perform cleanup. It has the same parameters as the `Initialize` method, however it does not return any value.

The BusEnumerator class

Classes which extend the `BusEnumerator` class, implement the actual logic of enumerating the bus devices. The `BusDriver` class uses such classes to create `DeviceFound` messages, which are sent over to the bus manager.

The `BusEnumerator` class defines an abstract method, called `EnumerateDevices`. This method is called by the `BusDriver` to generate a sequence of messages, which describe the devices connected to the bus. However, the method does not return any value. The XML messages are built by calling special helper methods of the `BusEnumerator` class. These methods are presented in listing 13.

When the enumerator discovers a device, it must begin a new XML message by calling the `StartPciConfig` or `StartPnpConfig` method, depending on what `IoConfig` object should represent the device in the I/O manager. Next the enumerator calls methods which add information about the device's resources to the XML message. Hardware ports can be added using one of the methods: `WriteIoPort`, `WriteIoMappedPort` or `WriteIoPortRange`. `IoDma` objects are declared using the `WriteIoDma` method, and so on. After all resources are added to the message, the bus enumerator must call `EndPciConfig` or `EndPnpConfig`. This method completes the XML message and after that the bus enumerator may start a new one.

The use of such API makes it easy for the `BusEnumerator` to perform its task easily and simply, without touching any XML. However, all methods presented in listing 13 are declared as `virtual`, which means that they can be overridden in subclasses. The underlying `XmlWriter` object can be obtained using the `GetWriter` method. This enables bus enumerators to change or fine-tune the actual XML which is passed over to the bus manager.

Moreover, the `BusEnumerator` class provides virtual methods to implement initialization and cleanup. The `InitializeEnumerator` is called before device enumeration. If the method returns `false` then the enumeration is not performed. After the enumeration is completed and the XML document containing device descriptions is ready, the `FinalizeEnumerator` method is called.

Listing 13: Methods of the BusEnumerator class for building XML messages which describe devices discovered by the bus driver

```
void StartPciConfig(string! type, uint identifier,
                    string! signature, string location);

void EndPciConfig();

void StartPnpConfig(string! type, string[] identifiers,
                    string! signature, string location);

void EndPnpConfig();

void StartRangeList();

void EndRangeList();

void WriteIoPort(string! name, ushort port, ushort size,
                 bool readable, bool writeable);

void WriteIoMappedPort(string! name, uint address, uint size,
                       bool readable, bool writeable);

void WriteIoPortRange(string! name, uint port, uint size,
                      bool readable, bool writeable)

void WriteIoDmaRange(string! name, int channel, int size);

void WriteIoIrqRange(string! name, byte line, byte size);

void WriteIoMemoryRange(string! name, UIntPtr addr, UIntPtr size,
                        bool readable, bool writeable);
```

There are also two methods which can be used to modify the beginning and end of the XML document which contains descriptions of the `IoConfig` objects sent to the bus manager. The `InitializeXmlDocument` method is called to write the document's root node. The method has a single parameter of the `XmlWriter` and it does not return any value. The default implementation creates a root XML node called `BusEnumeration`. After the device enumeration is complete, the `FinalizeXmlDocument` method is called. Similarly to `InitializeXmlDocument`, the method has a single parameter of the `XmlWriter` and it does not return any value.

3.3.3. Example – The PCI Bus Driver

This subsection presents the new PCI bus driver as an example of using the Bus Driver Helper Library to build a SIP-based bus driver.

First of all, the driver must implement the `DriverCategoryDeclaration` class, which is presented in listing 14. The driver supports devices with signatures `/pnp/PNP0A03` and `/pnp/PNP0A08`. It defines a single port range for the PCI port and a standard channel service provider channel. The bus control channel is not specified here, because it is not a service provider contract channel. It will be automatically binded by the Bus Driver Helper Library and we will access its endpoint later.

Listing 14: The DriverCategoryDeclaration subclass for the PCI bus driver

```
[DriverCategory]
[Signature("/pnp/PNP0A03")]
[Signature("/pnp/PNP0A08")]
internal class PciBusResources : DriverCategoryDeclaration
{
   [IoFixedPortRange(Base = 0x0cf8, Length = 0x08, Shared = true)]
   internal IoPortRange configPort;

   [ServiceEndpoint(typeof(BusContract))]
   internal TRef<ServiceProviderContract.Exp:Start> sp;

   internal int DriverMain(string instance)
   {
      // Verify resources
      ...

      ServiceProviderContract.Exp sp = ((!)sp).Acquire();
      try {
         PciBusDriver driver = new PciBusDriver();
         driver.Run(this, sp);
      }
      finally {
         delete sp;
      }

      return 0;
   }
}
```

The `DriverMain` method, which is the entry point of the driver application, needs to verify that all required resources are present or exit the application otherwise. Next, the application acquires the service provider channel endpoint, creates a `PciBusDriver` and invokes its `Run` method. This method implements the main `switch receive` statement on the bus control channel and when it finishes its execution, the driver's process is terminated.

The `BusDriver` subclass, which is presented in listing 15, is very simple in case of the PCI bus driver. It only implements a parameterless constructor which automatically specifies `PciBus` as the name of the driver and overrides the abstract `CreateEnumerator` method and returns an instance of the `PciEnumerator` class – the `BusEnumerator` class implementation for the PCI bus driver.

The `BusEnumerator` subclass for the PCI driver is presented in listing 16 (some of the code was omitted to minimize the size of the listing).

Fist of all, the class implements a constructor with a `IoPortRange` parameter, which stores the PCI address and data port in local variables.

Next, the abstract `EnumeratedDevices` method is overridden, to implement enumerating the PCI bus. The method loops through all available PCI device identifiers (lines 14-15) and tries to find a device for each identifier (using the PCI ports stored by the constructor). If a device is found then a signature is generated for the device (line 23) and the proper type of the `IoConfig` object which will represent the device is determined (lines 24-31).

Finally, the helper methods from the `BusEnumerator` superclass are used to generate an

Listing 15: The BusDriver subclass for the PCI bus driver

```
internal class PciBusDriver : BusDriver
{
   public PciBusDriver() : base("PciBus")
   {
   }

   protected override BusEnumerator! CreateEnumerator(
      DriverCategoryDeclaration! resources)
   {
      return new PciEnumerator(pciResources.configPort);
   }
}
```

XML document for the device (lines 34-39). The `StartPciConfig` method is used to generate an XML node representing a `PciConfig` object. Then we write out the address and data ports for the PCI device, which will be nested in the root XML node. Finally, the `EndPciConfig` method is used to close the XML node representing the PCI device.

That's all what the enumerate method must do. The Bus Driver Helper Library will automatically create a `DeviceFound` message and send it over the bus control channel to the bus manager.

3.4. Tests

This section presents results of tests which were performed to compare the performance and reliability of the bus device drivers using the new bus driver model with the drivers using the old kernel-based model.

Due to lack of compatible hardware, I was not able to run Singularity in a non-virtual environment. Therefore, for testing purposes Singularity ran in a virtual machine on Microsoft Virtual PC. The host system was Microsoft Windows 7 Ultimate, running on an Intel$^{®}$ Core^{TM} i7 CPU (1.73 GHz) with a Mobile Intel PM55 chipset and 4 GB of DDR3 RAM (1333 MHz). The virtual machine was configured to use 512 MB of RAM.

Performance tests were run against the initialization procedure of Singularity's I/O subsystem. In the kernel-based model this procedure includes enumerating available devices and loading their drivers. In the new SIP-based model the procedure also includes initialization of the bus manager, starting driver SIPs and all channel-based communication between bus drivers and the kernel. All measurements were made using the `Microsoft.Singularity.PerfSnap` type, which accesses usage statistics gathered by the Hardware Abstraction Layer and various kernel structures (such as the Memory Manager).

The IDE bus driver executes a lot longer than the PCI bus driver and most of the time is spent on initializing and loading IDE disks. That's why tests were performed on two configurations of the I/O subsystem. The first one uses all bus drivers and the second one does not use the IDE driver.

3.4.1. CPU Usage and Execution Time

The first test was performed to measure the number of CPU cycles and execution time of Singularity I/O subsystem's initialization procedure. The SIP-based model implementation introduces many new threads, which need to synchronize with each other. This may cause

Listing 16: The BusEnumerator subclass for the PCI bus driver

```
internal class PciEnumerator : BusEnumerator
{
   private IoPort! addr;
   private IoPort! data;

   public PciEnumerator(IoPortRange! ports) : base("pci")
   {
      addr = (!)ports.PortAtOffset(0, 4, Access.Write);
      data = (!)ports.PortAtOffset(4, 4, Access.ReadWrite);
   }

   protected override void EnumerateDevices()
   {
      for (uint bus = 0; bus < MAXBUSES; bus++) {
         for (uint device = 0; device < MAXDEVICES; device++) {
            uint identifier = IdentifierFromUnits(bus, device, 0);
            uint u = Read32(identifier, 0);
            // Try to find device with id: identifier
            ...
            if (u == ~0u || u == 0) continue;

            hadDevices = true;
            string signature = ...
            string configType = "Microsoft.Singularity.Io.";
            switch (u & PciConfig.PCI_TYPE_MASK) {
               case PciConfig.PCI_DEVICE_TYPE:
                  configType += "PciDeviceConfig"; break;
               ...
               default:
                  hadDevices = false; break;
            }

            if (hadDevices) {
               StartPciConfig(configType, identifier, signature, null);
               WriteIoPort("AddressPort", addr.Port, addr.Size,
                           addr.Readable, addr.Writable);
               WriteIoPort("DataPort", data.Port, data.Size,
                           data.Readable, data.Writable);
               EndPciConfig();
            }
         }

         if (!hadDevices) break;
      }
   }
}
```

an increase in execution time without increasing the number of CPU cycles. Additionally, execution time depends on hardware latencies. Therefore, both measures may indicate different results.

The test was performed ten times for each system configuration to analyze the stability of the results. The CPU usage during initialization of the I/O subsystem for ten system startups is presented in figure 3.2 and the execution times are presented in figure 3.3. Analogous results for the configuration without the IDE bus driver are presented in figure 3.4 and 3.5[1]. The table in figure 3.6 summarizes the results of the tests (CPU usage is presented in clock cycles and time is presented in nanoseconds).

The tests indicate that the new driver model introduces a slight overhead, which increases CPU usage, as well as execution time. The overhead is larger in case of the configuration without the IDE driver, however, the results are much more variable in that configuration compared to the configuration using all drivers.

3.4.2. Memory Usage

The next test measured memory usage of the new implementation in comparison to the kernel-based model. The amount of memory used by the initialization procedure is deterministic and is always the same for each system startup. Furthermore, the amount of used memory does not depend on the total amount of memory available to the system.

Figure 3.7 presents the total number of bytes allocated during initialization of Singularity's I/O subsystem[2]. The tests indicate a significant increase in memory usage in the new implementation. The configuration with all bus drivers required 74.33% more memory than the kernel-based implementation and the configuration without the IDE driver needed 54.64% more memory. The reason for this fact is the necessity to allocate memory for the driver's SIPs and memory needed to build the XML-based messages containing device descriptions. Maybe some of this additional memory would be released by the garbage collector when free memory runs low in the system, however none of the performed tests proved this case.

However, performance was not the main goal of the new bus driver model implementation and it can probably be well optimized, if required. Nevertheless, an increase in memory usage by a couple of megabytes should not be an issue on modern computer systems.

3.4.3. Crash Test

The final test was performed to check the reliability of the new SIP-based bus drivers compared to the kernel-based drivers. The test was performed by implementing a failure during device enumeration in the PCI bus driver. This simulates quite accurately an unexpected failure of the driver.

Predictably, the faulty kernel-based driver caused a failure which halted the kernel and stopped the system from booting successfully. On the other hand, the faulty SIP-based driver caused the driver's process to end unexpectedly, however the system booted successfully. Moreover, in the latter case, the PCI devices which were detected before the failure, were properly registered in the system.

This test seems to prove, that SIP-based drivers greatly enhance system stability and reliability. Especially, this concerns drivers for modern bus devices, which are more prone to failures due to their complexity.

[1]Note that the system startups in this test are distinct from the startups presented in the previous test.

[2]The test takes into account the number of bytes which were released during the initialization of the I/O subsystem.

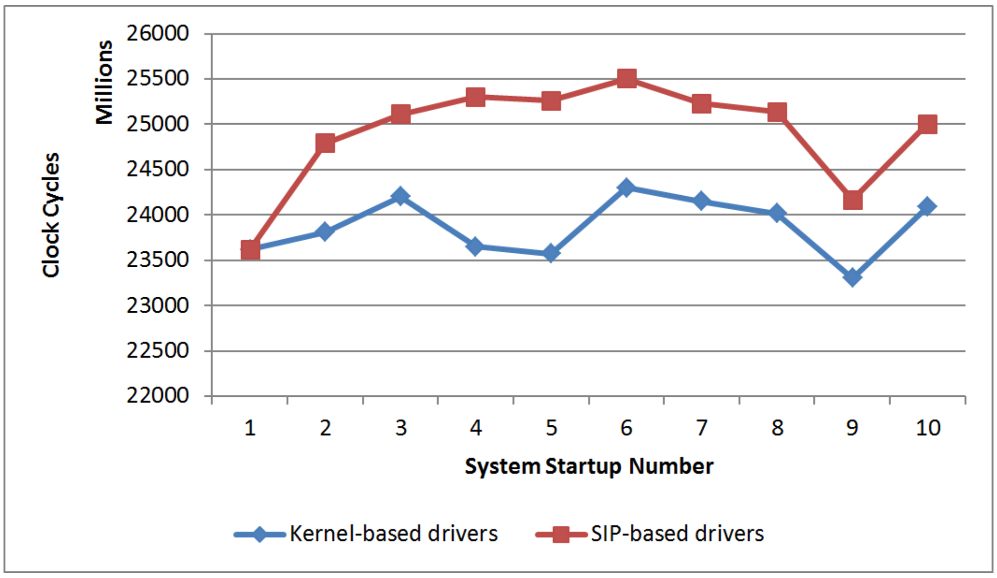

Figure 3.2: CPU usage during initialization of the I/O subsystem with all bus drivers

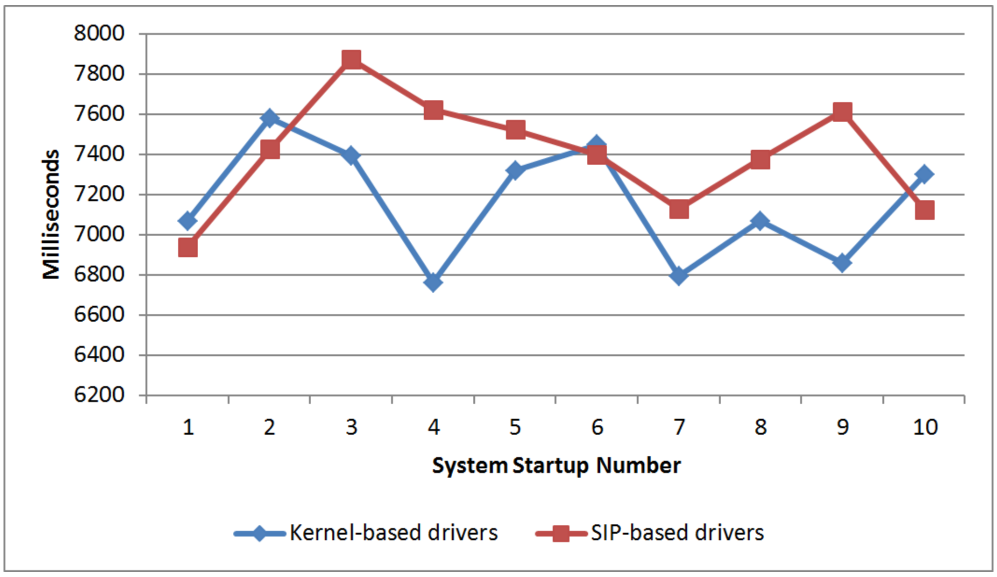

Figure 3.3: Execution times during initialization of the I/O subsystem with all bus drivers

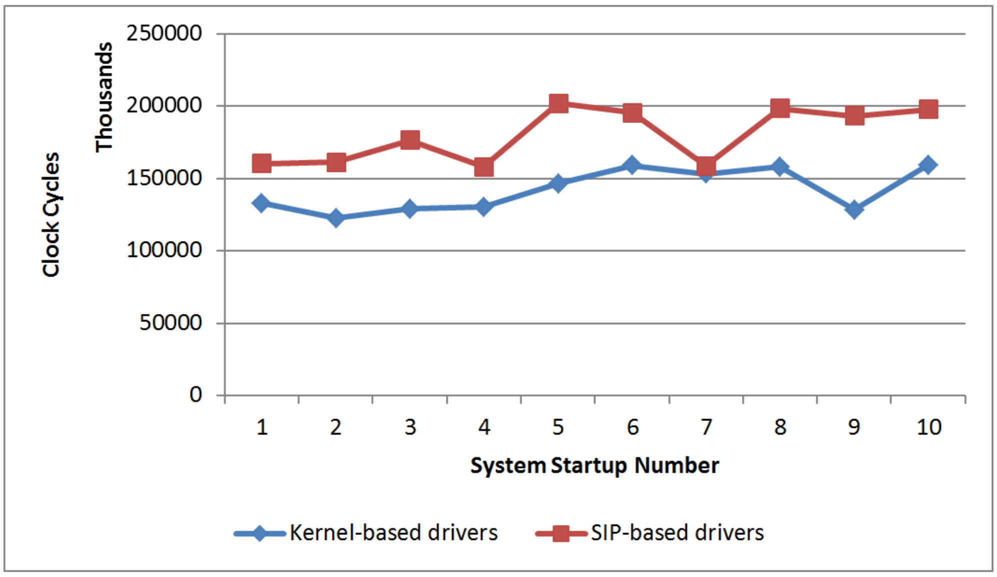

Figure 3.4: CPU usage during initialization of the I/O subsystem without the IDE bus driver

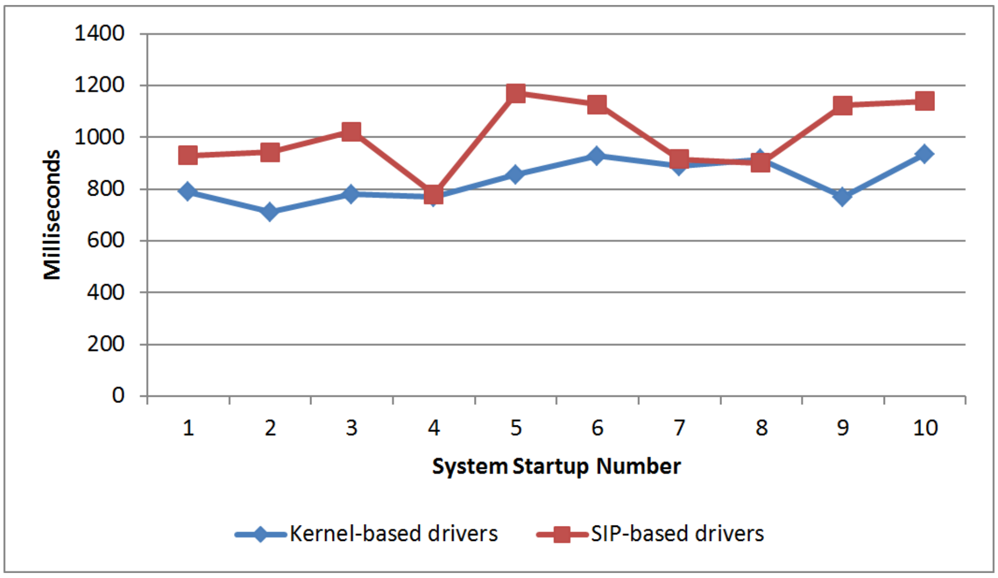

Figure 3.5: Execution times during initialization of the I/O subsystem without the IDE bus driver

	Kernel-based drivers (all drivers)	SIP-based drivers (all drivers)	Kernel-based drivers (no IDE)	SIP-based drivers (no IDE)
CPU				
Average	23872162526	24913202158	141785232,6	180074410,6
Overhead	-	4,36%	-	27,01%
Standard Deviation	328084164	582865626	14684426,6	18888686,55
AVG / STDEV	1,37%	2,34%	10,36%	10,49%
Execution Time				
Average	7158430,7	7400911,4	833639,7	1005339,1
Overhead	-	3,28%	-	17,08%
Standard Deviation	290061,7326	278580,0033	79688,71977	130594,647
AVG / STDEV	4,05%	3,76%	9,56%	12,99%

Figure 3.6: Summary of CPU usage and execution time tests

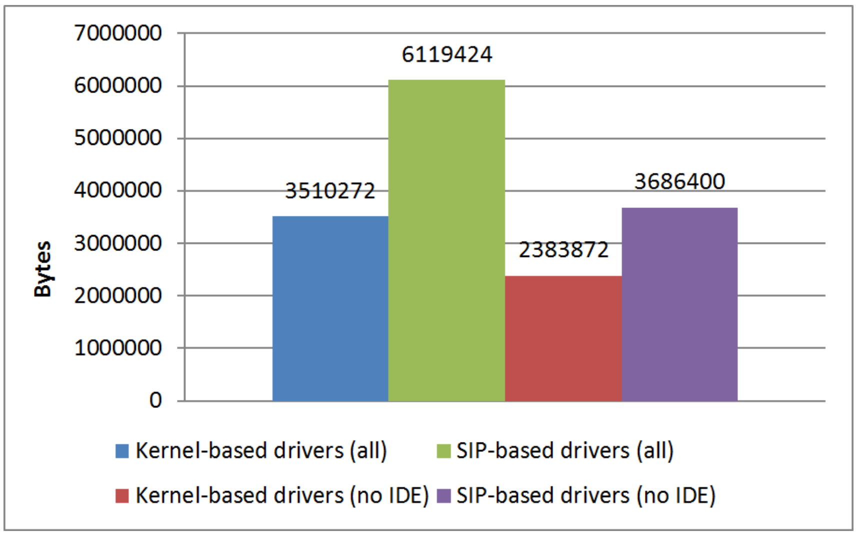

Figure 3.7: Number of bytes allocated during initialization of the I/O subsystem

Chapter 4

Conclusions

Isolating device drivers in separate processes leads to a better system reliability and stability. The operating system is protected from bugs in driver code and drivers may be sandboxed more easily. Moreover, in many cases, it should be possible to restart crashed drivers without restarting the entire operating system. However, implementing process-based device drivers is not a simple task. It requires specific system mechanisms and cannot be generally done on any operating system. Furthermore, implementing process-based device drivers requires more work from programmers, since process-oriented driver models are more complicated then the kernel-oriented ones. Despite the mentioned difficulties, in specific operating systems, such as Singularity, implementing process-based driver models is completely attainable and beneficial.

It seems that for some low-level devices, implementing process-based drivers does not make much sense and this also concerns low-level buses, such as the Basic Input/Output System. On the other hand, complex hardware such as graphic adapters, storage controllers, etc. highly benefit from the process-based driver model in terms of crash resistance.

During my work on this thesis, I designed and implemented a new bus driver model, which supports process-based bus device drivers for Singularity. To minimize the amount of work needed to create new bus drivers, I developed a new helper library which implements most of the functionality required to implement bus drivers using the new model. This library avoids code duplication and takes advantage of many high-level, object-oriented programming language constructs. Additionally, I analyzed the source code of Singularity's I/O subsystem and described the subsystem in this thesis.

In the case of Singularity, implementing process-based bus drivers is more complicated than implementing regular isolated drivers, due to their tight interconnection with the I/O subsystem. However, the bus driver model presented in this thesis enables implementing SIP-based drivers in Singularity for most bus devices. Tests showed that the overhead introduced by this more complex driver model is acceptable. Furthermore, thanks to the Bus Driver Helper Library, the amount of work required from device driver programmers is minimized.

While working on this thesis, I encountered many problems involving the design of the new bus driver model, as well as its implementation. Designing the presented driver model required a detailed understanding of Singularity concepts, such as channels and contracts. During the design of the driver model, I had many ideas which did not turn to work out (for example implementing I/O resource objects as rep structs). Later, during implementation of the driver model and new drivers, I encountered many problems concerning details of the I/O subsystem. Lack of documentation was one of the major difficulties.

This thesis will be sent to the team which currently maintains the Singularity project at Microsoft. Hopefully, the thesis will be published on Singularity's website and it will serve as a documentation of Singularity's I/O subsystem for future developers working on the project. I hope it will be useful for Singularity driver developers, and especially bus driver developers.

4.1. Future Work

There are still some areas of research associated with the issues discussed in this thesis, which might be addressed in the future.

Most of all, drivers for dynamically attached hardware, such as USB devices, are well worth considering in context of Singularity. Implementing such drivers would enable further verification of the developed SIP-based driver models and could find potential problems with their design. This might not be a simple task, because it requires implementing drivers for complex bus devices (for instance the USB bus) and extending Singularity's I/O subsystem to support loading drivers and registering hardware not only during startup, but also later when the system is running.

Another minor issue is the current implementation of registering available bus drivers, which is done statically in kernel code. It would make sense to implement some mechanism which would automatically detect installed bus drivers and register them in the system. This would probably be done best by extending the manifests of the bus drivers.

Bibliography

[1] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte, *The Spec# Programming System: An Overview*, *Proceedings of Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*, Springer Verlag, Marseille, France, 2004, http://research.microsoft.com/en-us/projects/specsharp/krml136.pdf

[2] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, Steven Levi, *Language Support for Fast and Reliable Message-based Communication in Singularity OS*, EuroSys 2006, Association for Computing Machinery, Inc., April 2006, http://research.microsoft.com/apps/pubs/default.aspx?id=67482

[3] Galen C. Hunt, James R. Larus, David Tarditi, and Ted Wobber, *Broad New OS Research: Challenges and Opportunities*, Microsoft Research, in Proceedings of Tenth Workshop on Hot Topics in Operating Systems (HotOs), USENIX, June 2005

[4] Galen C. Hunt, James R. Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, Brian D. Zill, *An Overview of the Singularity Project*, Microsoft Research, October 2005, http://research.microsoft.com/pubs/52716/tr-2005-135.pdf

[5] Galen C. Hunt, James R. Larus, *Singularity: Rethinking the Software Stack*, ACM SIGOPS Operating Systems Review, no. 41/2, page 37-49, Association for Computing Machinery, Inc., April 2007, http://research.microsoft.com/apps/pubs/default.aspx?id=69431

[6] Michael B. Jones, Joseph S. Barrera III, Alessandro Forin, Paul J. Leach, Daniela Rosu, Marcel-Catalin Rosu, *An Overview of the Rialto Real-Time Architecture*, Microsoft Research, July 1996, http://www.barrera.org/rialto/rialto.htm

[7] Michael Spear, Tom Roeder, Orion Hodson, Galen C. Hunt, Steven Levi, *Solving the Starting Problem: Device Drivers as Self-Describing Artifacts*, EuroSys 2006, Association for Computing Machinery, Inc., March 2006, http://research.microsoft.com/apps/pubs/default.aspx?id=69432

[8] Dachuan Yu, Andrew J. Kennedy, and Don Syme, *Formalization of Generics for the .NET Common Language Runtime*, Microsoft Research, January 2004, http://research.microsoft.com/pubs/64032/formalizationofgenerics.pdf

[9] *Attributes Overview*, Microsoft Developer Network http://msdn.microsoft.com/en-us/library/xtwkdas5%28VS.90%29.aspx

Appendix A

CD Contents

A CD has been attached to this thesis. The contents of the folders on that CD is described in this appendix.

- **thesis/** – contains the source and graphic files of this document (in LaTeXformat) and test results,
- **source/rdk.zip** – contains source files of the original Singularity OS (Research Development Kit 2.0 version),
- **source/impl.zip** – contains source files of the extended Singularity OS with SIP-based bus drivers,
- **iso/rdk-all.iso** – compiled binary version of the Singularity RDK 2.0 with kernel-based bus drivers,
- **iso/rdk-noide.iso** – compiled binary version of the Singularity RDK 2.0 with kernel-base bus drivers (IDE driver disabled),
- **iso/impl-all.iso** – compiled binary version of the extended Singularity OS with SIP-based bus drivers,
- **iso/impl-noide.iso** – compiled binary version of the extended Singularity OS with SIP-based bus drivers (IDE driver disabled)
- **docs/license.txt** – Singularity RDK license,
- **docs/Building and Running Singularity RDK 2.0.pdf** – information about building, running and debugging Singularity on Virtual PC.