Debug Interfaces for Modern Operating Systems

Niklas Quarfot Nielsen

Kongens Lyngby 2010
Technical University of Denmark
DTU Informatics
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk
Summary

Defects in software are unavoidable. These defects make system fail every day all over the world. These failures can vary from just being an annoyance for the user to be life critical. For example a non-responsive media player or web browser compared to malfunctioning space shuttle software. The discovery, tracing and fixing of defects in software can be a very tedious and time consuming task. It is estimated that at least half of the development time used in a software project, is used for tracing and removing bugs. A process which includes *debugging*. Debugging of operating systems can be an even more difficult and tedious task to overcome than debugging traditional software. Operating systems is a category of software, which interacts directly with the unforgiving environment of the hardware. It must protect itself against misbehaving user processes and malfunctioning hardware. Operating systems is also one of the most complicated pieces of software which runs on a computer. This level of complexity combined with the harsh environment of the hardware makes it a very difficult job to debug operating systems. This thesis will investigate the possibilities for kernel debugging, user space debugging and postmortem analysis of kernel crashes.
This report is written as a bachelor thesis at Technical University of Denmark (DTU). Readers should at least have knowledge which corresponds to a DTU computer science student at his/hers 6th semester to get full benefit from the report. Readers are not expected to be confident with the domain of operating systems, software bugs, hardware support for debugging or user space debugging. This is covered in the report however, prior knowledge to this domain will make it easier to read and understand.

Enjoy

Lyngby, June 2010

Niklas Quarfot Nielsen
First of all, I want to thank Sven Karlsson for being advisor on this thesis. He has been a great help in the entire phase and has been tolerant and understanding. He has provided constructive feedback during the report writing which has helped bringing it to its current level.

Thanks to my good friend Edward A. Cerullo, for helping me with proof reading. His attempt to teach me how to write reports is admirable.

Thanks to another good friend, Bo Stendahl Sørensen. He has also provided constructive feedback on the report. He did his thesis 6 months ago and has experience regarding report writing at DTU.

Without you, this thesis would not have been what it is today.
Abstract

Defects in software are unavoidable. These can be very difficult to trace as defects leaves infections in the program state which may spread. These infections may under certain circumstances trigger a fault, but may be hidden for a very long time or never be discovered. The number of defects and their severity affects the quality of software, as reliability and correctness must be a minimum requirement. Tools and techniques to prevent or limit the amount and severity of these defects are wanted.

Operating systems are very complex software that interacts directly with the unforgiving environment of the hardware. It must protect itself against misbehaving user processes and defects in hardware. This level of complexity along with the unforgiving environment makes a difficult task to instrument and examine its execution and state. FenixOS is an operating system being developed at the Technical University of Denmark and does not incorporate any debugging infrastructure at this point in time. The possibilities for such a framework have been investigated. A debug infrastructure which provides remote debugging facilities, debugging of user space processes and facilities to report useful information in case of system crash has been chosen for design and implementation. This gives a sufficient set of primitives for a kernel developer to ease the process of debugging operating systems. This prototype implementation can be the foundation of a later more complete debugging infrastructure.
Contents

Summary i
Preface iii
Acknowledgements v
Abstract vii

1 Introduction 1
1.1 Problem description . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

2 Theory 5
2.1 Software defects, infection and failure - bugs . . . . . . . . . . . . . . . 5
2.2 Debugging . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.3 Operating systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.4 Bugs in operating systems . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.5 Debugging operating systems . . . . . . . . . . . . . . . . . . . . . . . 14

3 Analysis 17
3.1 Hardware support for debugging . . . . . . . . . . . . . . . . . . . . . 17
3.2 Remote debugging . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.3 Debugging interfaces for user level processes . . . . . . . . . . . . . . 29
3.4 Postmortem analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

4 Design 35
4.1 Problem statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.2 Design proposals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.3 Debug infrastructure design . . . . . . . . . . . . . . . . . . . . . . . . 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4 Project management</td>
<td>41</td>
</tr>
<tr>
<td>5 Implementation</td>
<td>47</td>
</tr>
<tr>
<td>5.1 Remote debugging</td>
<td>47</td>
</tr>
<tr>
<td>5.2 User space debugging</td>
<td>52</td>
</tr>
<tr>
<td>5.3 Postmortem analysis</td>
<td>55</td>
</tr>
<tr>
<td>6 Evaluation</td>
<td>57</td>
</tr>
<tr>
<td>6.1 Remote debugging</td>
<td>57</td>
</tr>
<tr>
<td>6.2 User space debugging</td>
<td>64</td>
</tr>
<tr>
<td>6.3 Kernel and user panics</td>
<td>67</td>
</tr>
<tr>
<td>6.4 Summary</td>
<td>69</td>
</tr>
<tr>
<td>7 Discussion</td>
<td>71</td>
</tr>
<tr>
<td>8 Conclusion</td>
<td>75</td>
</tr>
</tbody>
</table>
Human error is unavoidable and this is unfortunately applicable to software development also. It is not necessarily because developers are lazy or inattentive, but because the complexity of software is at such a high level the human brain can not fully manage it [Pan, 1999].

When a defect has been introduced to the code by a developer, it may trigger a fault under certain conditions. A common term for this is *software bug*.

At least half of total development time for an average software project is used on locating and removing bugs with testing and debugging [Beizer, 1990]; a process which is often difficult and tedious. It is an important phase in a development cycle, as the amount of bugs and their severity affects the quality and reliability of the final product. In the end, this can determine whether a customer chooses your product or not.

Operating systems are a category of software that interact directly with the hardware of modern computer systems. They are often very complex and they provide the ability to connect hundreds of different kinds of peripherals. This support are one of the reasons for the high complexity of operating systems and they often consist of millions of lines of code. This level of complexity, along with the unforgiving environment of the hardware, make it an even more tedious and time consuming task to debug operating systems than traditional application software.
Tools and frameworks can be provided to ease the process of debugging software. They can cut down the time needed for tracing, locating and fixing bugs.

FenixOS is a new operating system which is being developed at the Technical University of Denmark. It aims to be a modern operating system prepared for the future developments in hardware for many core systems. Presently, FenixOS does not provide any debugging facilities and this complicates development of the operating system.

This thesis will investigate how debugging in operating systems is done, and propose and motivate a debugging infrastructure for FenixOS.

1.1 Problem description

A debug infrastructure is needed for FenixOS, as none is currently available. This debug infrastructure must be a part of the kernel which runs directly on top of the hardware. To control execution of running processes and the kernel, processor manufactures must provide a set of functionalities for debugging support. These functionalities must be investigated to know what can be provided in the debug infrastructure.

Operating systems are responsible for operation of the entire hardware system. When debugging operating systems, debuggers running on the same machine as the one being debugged may affect its current state. This is because screen and keyboard drivers may change the state of the kernel, which may leave data examination untrustworthy. For this reason, the ability to remotely debug FenixOS and the options available for remote debugging should be investigated.

A user space debugger is a special process that has the ability to spawn or hook into a debugged process. The debugger can control execution and examine the states of the debugged processes. For this to be possible, the underlying operating system must provide the ability for the debugger to read the memory of another process and control its execution. This should be done in a controlled manner, as normal processes should not be able to interferer with each other. User space debugging should be investigated to provide the functionality required.

Operating systems may fail with no indication of error and leave the system in a useless state. To trace and locate a defect in FenixOS, a system is needed to give enough information to tell where and why the system crashed. Postmortem techniques for analyzing kernel crashes should be investigated.
In summary, the following subjects should be investigated.

- Hardware support for debugging
- Remote debugging
- Debugging interfaces for user level code
- Postmortem analysis of kernel crashes

Based on knowledge gathered, a debugging infrastructure for FenixOS should be presented and a prototype implemented.

1.2 Structure

The report is structured as follows:

- **Introduction** defines the problem this thesis will investigate. To understand the thesis, this chapter should be read by all.
- **Theory** covers the background material needed to understand the core concepts of operating systems and debugging. Advanced readers who knows the domain can skip this chapter.
- **Analysis** lists the alternatives and options for solving the problem defined in the problem description. This chapter takes decisions based on listed argumentation and it should be read by all to understand the design chosen.
- **Design** will be proposed, based on the analysis. This includes overall design along with more detailed design of subsystems. To understand rest of the report, this chapter should be read by all as new terms and definitions are mentioned.
- **Implementation** will contain specific details on the implementation of the design proposed. This chapter can be skipped if the reader does not want to delve into technical details.
- **Evaluation** will contain test results which should argue for the correctness of the chosen design and the implemented code. To determine the level of success of the thesis, this chapter should be read by all.
- **Discussion** will discuss the system implemented. What it lacks and how it should be improved, and what limitations it has.
- **Conclusion** will be a summary and an evaluation of the entire thesis. Compare this to the introduction to get a better overview of the success of the thesis.
This chapter will cover the background needed to understand the domain of this thesis. The different sections can be read apart from each other or left out, if the reader is already confident with the domain of operating systems, software bugs and debugging.

2.1 Software defects, infection and failure - bugs

Human error is unavoidable and this is no different for software development.

Errors or flaws in software are often called defects, which originates from the latin word defectus meaning weakened or lacked. Software defects are detected when activated and this can be experienced as system misbehavior or failure. The severity of these failures can vary from being nonsevere i.e. just being an annoyance to the user, to be very severe i.e. life critical in for example medical, aerospace or military equipment.

An unfortunately long list of software related failures exists which includes loss of hundreds of millions of dollars or cause of death [Huckle, 2010]. For example: The radiation machine Therac-25 killed 6 patients because of radiation overdose of 100 times the normal radiation caused by a overflow flaw in the controlling software [Baase, 2008].
Software defects are a major hindrance for the success of a software project. Some defects may even lead to projects being completely dismissed if the defects are unrecoverable or too expensive to fix. The quality of a product is tightly bound to the amount of defects it contains. Quality is usually a measure for a product's ability to fulfill its specification and requirements. A minimum requirement must be that the product is correct and reliable. This is a requirement which does not necessarily hold if a piece of software contains many and/or severe defects.

How are these defects introduced? Initially a programmer writes code which are transformed into an executing program by a compiler. Defects in software must initially be introduced by the programmer who wrote the software or the tool which compiled the executing program. It is usually not the intention of a developer to make defects in his or her software, however the complexity level of software makes it extremely difficult to fully manage.

A defect in a piece of software infects the state of it, i.e. puts it in a state the developer did not expect. This defect may spread around the program and put multiple regions in an infected state. This is illustrated in figure 2.1.

These infected states may cause the system to fail, produce erroneous results or behave in an unwanted manner. A defect may be activated only under certain circumstances. This defect may even be activated and not be discovered. This, along with the spread of infected states in a program, complicates the tracing and correction of the defect tremendously. The famous quote by Dijkstra holds; "Testing shows the presence, not the absence of bugs".

A common word for software defect, its infection of the program state and failure is bug. The term bug is used in many contexts and is a loose term but it is widely used. In the rest of the thesis, bug will be used as a common word for defect, infection or failure.
2.1 Software defects, infection and failure - bugs

Figure 2.1: Defect infects the state of the program and spreads
2.2 Debugging

The process of eliminating software defects or limiting the amount of them is often called debugging i.e. removing bugs.

As argued in the previous section, software defects are highly unwanted as they can cause annoyance, damage or even death. But how are software defects fixed?

There exists multiple options to help reducing the amount bugs. Some options worth mentioning are:

- Programming language features
- Static code analysis
- Verification
- Software engineering
- Instrumentation

Programming languages often offers features to limit the amount of bugs a developer introduces. Type checking may prevent foolish assignments and calculations, for example multiplying a string with an integer. A language like Ada provides custom types such as a type for inches and a type for centimeters. These can not be mixed in any other way, before they have been converted into the same type. Some languages provide a way of isolating different parts of code from each other in more or less closed domains, which limits accidental use of unwanted procedures and variables. In a programming language like C++, this is done with abstract containers called namespaces. Syntactical errors and obvious logical errors are often detected by the compiler. Compilers transforms source code into executable binary files.

Static code analysis is a process where an analysis tool runs through the source code of a program and reports potential dangerous situations before any compilation takes place. Situations such as errors in logic where a branch or a portion of code may never be executed because of incorrect logic, is one example. Errors in memory where a null pointer exception caused by a pointer that was dereferenced before it was initialized to a memory address is another example. These tools can help companies avoid shipping of products with potential flaws, and this makes static code analysis a large industry. These tools may prevent some flaws in the code, but it can not tell what, where or when software went wrong when software does not act as it was supposed to do.
2.2 Debugging

Verification exists to investigate the state space of a program, ensuring that no state violates the properties of the program. This is the most thorough method of detecting defects, however when programs grow in complexity (variables and transitions to change these), verification can become inapplicable as the state space can become enormous. Therefore, simplified models of the software are often verified instead. This is required for software in some very safety critical systems. Evaluation Assurance Level (EAL) is a standard for evaluating the level of reliability and security of a system regarding to the level of testing and verification. Software with the highest EAL must be formally verified in full.

Proper software engineering can reduce common types of bugs, which originate from misunderstanding or misinterpretation between developers. Mixing of programming styles regarding naming and logic, may introduce bugs which are hard to discover. Software engineering can standardise this and also ensure that software will follow the given specification. Programming style may assure that consistency checking is used. This can be done using assertions, which acts like a predicate or invariant. If a given condition is not true or valid, execution stops and the incident gets reported. This ensures the program is in a sane state.

A common applicable method for locating and fixing bugs is instrumentation. Instrumentation is a commonly used engineering practice for measuring and controlling systems. This involves measuring of variables such as pressure or temperature and means to control such variables such as valves and circuit breaks. In computer science perspective, measure and control is usually of the state of the software.

When trying to fix a bug, reproducibility is a key issue. To investigate the cause of a failure, it must be possible to reproduce the conditions that lead to it. If it is not to be reproduced, it is hard, if not impossible, to show the presence of the failure.

When the failure can be reproduced, the possible origins of the failure can be located. These should be isolated and investigated to locate the true origin of the failure.

There exists many techniques to do this, but can roughly be categorized into:

- Tracing execution / Observing
- Controlling execution / Examining
- Postmortem analysis of failed execution / Investigating

Tracing software is an effective way of observing the path the program follows to get to a certain state and to narrow the problem to a specific place in the code. It can be done with simple `printf` statements or more advanced frameworks can be applied to generate a standardized tracing. [Lehey, 2005].
Controlling execution gives the developer a strong tool to where he or she can:

- stop execution at breakpoints and examine the current state by looking at variable and data structure contents.
- step through the program per instruction, source code line or function basis.

Both primitives allow the developer to follow the execution and the state changes the program makes. Unfortunately, this often has major influence compared to normal execution with regard to timing. Single stepping slows execution tremendously. This can be a very large issue when debugging concurrent processes, as timing may differ when debugging as compared to running in production. This can complicate the tracing of synchronization related defects such as deadlocks and race conditions.

Postmortem analysis is a method for investigating the cause of a failure. Software can leave a set of information related to its state when it failed. This can be used to backtrack the problem and provide necessary information to reproduce the failure and isolate the defect.

### 2.3 Operating systems

An operating system is in general a piece of software which provides an abstraction for all underlying hardware and provides an forgiving environment for programs to exist. There exists many kinds of operating systems with different usage, design methodologies and philosophies. Usually, an operating system consists of a kernel which serves user processes and distributes resources such as CPU time, memory, access to hardware, etc., to these user processes. An illustration of this is presented in figure 2.2.

To prevent user processes from corrupting the system when they fail, privilege levels are often provided by the processor architecture. The result of the corruption could be changing memory of other processes or even in the kernel. Privilege levels creates a split between software running in privileged mode (supervisor mode) and in less privileged mode (user mode) and this split is a central issue in the design mythology of an operating system. Operating systems are often categorized into one of the following:

- Monolithic kernel
- Micro kernel
A monolithic kernel is one large program running in supervisor mode. All kernel related operations are carried out in the kernel i.e. nothing kernel related runs in user mode servers. An illustration of this structure is presented in figure 2.3.

This is the most simple approach but it has some downsides. If anything inside the kernel fails, the entire kernel fails as no surrounding environment can recover. The strengths of a monolithic kernels are that it is simple and can be fast, as a minimal number of context switches are needed to carry out system calls. System calls are the way for a user process to interact with the kernel. It works like a ordinary procedure call, but the privilege level is changed and the kernel context is loaded. In a virtual memory enabled system, the Memory Management Unit (MMU) needs to reload the address space when changing from user space to kernel space, and from one user process to another user process. This along with saving CPU registers and other state related variables are saved in a context switch. This is an expensive operation.

A micro kernel uses a different approach. As the amount of code increases, the amount of potential bugs increases [Lipow, 1982]. By having a minimal piece of software running in supervisor mode which only provide the most fundamental functionality and moving as much as possible into user space processes, the ideas is that the overall
stability and security of the system should improve. The down side is the complexity, as the kernel related user space processes (called servers) needs to communicate between each other and with the kernel, reproducing and debugging bugs becomes a very difficult task. Performance can also become an issue as the amount of needed context switches increases tremendously when kernel operations should be carried out.

An exo kernel provides a thin layer between user processes and hardware, and it is even able to provide direct access to hardware, only taking care of isolating the processes from each other. This is a less common design mythology and will not be covered in depth.
2.3 Operating systems

2.3.1 FenixOS

FenixOS is a modern operating system developed at Technical University of Denmark, which targets future multi-core systems, and faces the problems these new parallel systems introduce.

FenixOS is a hybrid kernel, combining the monolithic and micro kernel design methodologies. The concept is to move as much as possible into servers running in user space, but put critical servers such as memory management into the kernel. This would remove some of the overhead from context switching to these servers. This loses some of the strength of the micro kernel, but if these critical servers fail, the system is not likely to be recoverable anyway.

![Hybrid kernel diagram](image)

Figure 2.5: Hybrid kernel. Bigger kernel layer than a micro kernel as critical tasks such as virtual memory management are handled inside the kernel. Tasks which are possible to move to user space space are done so.

FenixOS is primarily written in C++ which offers the object oriented paradigm for kernel development. This is a great strength, as operating systems tend to grow into very large and complex systems and C++ has the ability to separate and isolate domains and provide facilities to reuse existing code.
2.4 Bugs in operating systems

Operating systems are one of the most complex pieces of software which run on a computer. They often consist of several million lines of code [Leemhuis, 2009]. They interact directly with hardware and must try to protect themselves against misbehaving hardware, faulty user programs and badly written hardware driver code - a very difficult task to fulfill unfortunately. An operating system is only surrounded by the hardware which is a rather unforgiving environment. When a fault happens, the system may just hang - leaving no indication of where the fault happened and what triggered it. This can make bugs in operating systems very severe. An operating system may deny service for the entire system in case of a breakdown, or even expose the system for exploitation to attackers.

Operating systems run in the most privileged level the hardware allows, and this means defects in operating system code may corrupt the state of all running processes and the kernel itself. Usually hardware provides some mechanisms to protect the kernel from being affected by misbehaving user processes and system libraries. This split was mentioned in the operating system section and is sometimes called a privilege level. It can also be referred to as being a certain ring level. Ring 0 (most privileged) or one of the outer rings (less privileged). This is illustrated in figure 2.6.

When a defect leads to a security vulnerability, this may make it possible for an attacker to execute code in the privilege level of the operating system. By that time, the attacker can do what he or she wants. Evil code like worms and viruses can exploit these security holes and use them to infect a computer and spread to other computers, requiring thousands of man hours to remove them and fix the computers.

When an operating system fails, the circumstances which lead to the failure may be very complicated to reproduce. External input such as network data parsing, some data sequences on a hard drive along with hundreds of other possible inputs may be very hard, if not impossible, to reproduce.

In summary, bugs in operating systems can be very difficult to discover and track, and they can be very severe.

2.5 Debugging operating systems

There exist various ways and techniques for debugging operating systems. As mentioned in the previous sections, one of the most time consuming tasks of debugging is tracing the defect and observing the conditions that lead to failure.
2.5 Debugging operating systems

Figure 2.6: Security rings. Ring 0 is the most privileged with no restrictions (Operating system). Rings 1 to 3 are restricted in the set of permitted instructions. These instructions may even have different effects depending on the ring number. Usually rings 1 to 2 are for system libraries and ring 3 is for user processes. A system may have additional rings or less rings.

As with traditional software, the listed debugging method in section 2.2 also apply to debugging of operating systems. Instrumenting operating systems differs from instrumenting traditional software, as the operating system usually is responsible for providing an environment for such. For this reason, instrumentation of operating systems should be covered a bit more in depth.

Controlling execution gives developers the ability to observe the execution of the program while stepping through the source code. This is a powerful tool as the exact point in source code where the program fails can be found and the path and conditions investigated. For operating systems it is usually not possible or wanted to control it self. After all where should the debugger run? Remote debugging is a way to provide this control, without the debugger interfering with the operating system execution. A downside controlling execution is that it may interfere with the normal execution anyway. Timing may be different which would lead to a successful debug run, but will lead to faults when being run without a debugger attached. This can especially be an issue on parallel systems which modern operating systems usually are. Another issue is the security aspect. When controlling the execution of an operating system, this often requires at least reading from the system memory and sometimes writing to it. If a system is accidentally made debuggable, an attacker would be able to attach a debugger and read confidential information like password hashes, encryption keys, install backdoors or even just bring the system down.

In-kernel debuggers are debugger which are invoked on the same system, as the one being debugged. Even though they may interfere with the state of the kernel,
sometimes they may provide the functionality necessary to stop execution at some point or in case of a system crash and examine the state of the system. An example of such a debugger is Linux Kernel debugger called *KDB*.

When an operating system fails, it usually leaves the system useless and it is very difficult to get information regarding the crash. The mentioned *panics* are a way of telling the user what happened and the state of the operating system when it crashed. These *panics* are limited to report the most important information to the screen or serial device. Another technique is to make a core dump, which is a image of the memory along with the state of the CPUs on the time of the crash. This can later be analyzed and the problem investigated remotely. This is called postmortem analysis.
Chapter 3

Analysis

The analysis chapter will investigate the stated problems from the problem description. Options available to solve a given problem should be covered and a specific option should be chosen. This option should be selected based on proper argumentation.

3.1 Hardware support for debugging

FenixOS is in its present state developed to run on the AMD64 architecture. This is a 64 bit architecture which extends the popular 32 bit x86 architecture with respect to address space, registers, security functionality and much more. A substantial part of the following is directly related to the AMD64 architecture and not necessary applicable in other architectures, even though some of the debugging facilities is backward compatible with the x86 architecture.

To control execution of an operating system, facilities must be provided by hardware. The main concern is breakpointing and single stepping, as register and memory access in the kernel is already possible without any extra facilities needed. On the AMD64 architecture, breakpointing can be done in 2 ways:
• Soft breakpoints. This is a technique where the instruction data on a given address is swapped with a special instruction. This instruction data is called an opcode. When execution reaches this instruction, an interrupt is generated and an appropriate breakpoint handler can install the original instruction and hand control to a debug client. This is illustrated in figure 3.1 and 3.2.

• Hard breakpoints. This is a feature provided by hardware where 4 breakpoint addresses can be installed in special debug registers (DR0, DR1, DR2 and DR3). When the CPU executes an instruction, accesses memory or an I/O port which matches the address in one of the debug registers, an interrupt is generated and the incident conditions written to debug register DR6. From this point an appropriate breakpoint handler determines which breakpoint was triggered and can hand control to a debug client. This is illustrated in figure 3.3.

Debug client is here meant as an arbitrary handler, which could be a remote debugging stub. This handler can invoke user space processes, record statistics, call a remote debugger etc.

Figure 3.1: Soft breakpoint installation

Figure 3.2: Soft breakpoint triggering
3.1 Hardware support for debugging

1. CPU executes instruction
2. Instruction address matches debug register and interrupt handler is called
   - Interrupt handler
     - May call remote debugger or other controlling facility
3. Execution continues

Figure 3.3: Hard breakpoint triggering

Soft breakpoints are not limited by the number of debug registers available on the architecture, however they require write access to the text segments of the kernel. Soft breakpoints are only able to trigger on instruction execution, which means that memory watches are not possible. Another issue is Copy-On-Write. Virtual memory systems provide the ability to share pages between processes. This makes it extremely fast for, for example, spawning a new process from an existing one. But the technologies require that when one of processes alter data in a shared page, it must be copied into a new local one, which the process can alter. Usually, text segments of a process do not change. So these types of pages are usually just shared among the processes of the same program. If soft breakpoints should be supported, Copy-On-Write functionality should be added for text segments by the virtual memory system.

Hard breakpoints are limited by the number of breakpoints which can be installed at one time, and additional logic for recycle breakpoints needs to be created if additional breakpoints are needed. Hard breakpoints are more configurable than soft breakpoints, as you can specify an area of addresses which should trigger, and specify the type of access like memory read and/or write.

There exists additional methods for triggering a debug exception and control execution; this is single stepping. Single stepping is supported by the AMD64 architecture and triggers a debug exception (#DB) after each instruction. When entering the debug handler, single stepping is disabled such that the debug handler is not single steppable.

In AMD64, an additional set of debugging and performance measuring facilities has been added in Model-specific registers (MSRs). For debugging, an additional way of single stepping has been added. This is control transfer single stepping and provides a more coarse-grained single stepping where only jumps, interrupts and other control transfers triggers a debug exception. This gives a quick way of locating the area
which a defect may reside. The control transfer methods that trigger this kind of single stepping are:

- Jumps (JMP), conditions jumps (Jcc) and conditions loops (LOOPcc)
- Subroutine calls (CALL) and return from these (RET)
- Interrupt (INT), exceptions and return from these (IRET)
- System calls (SYSCALL and SYSENTER) and return from these (SYSRET and SYSEXIT)

A new recording facility tracks the set of information which describes what happened when the debug exception was triggered. The four registers are maintained by the processor and are listed below:

- Last branch to instruction pointer (LastBranchToIP)
- Last branch from instruction pointer (LastBranchFromIP)
- Last exception to instruction pointer (LastExceptionToIP)
- Last exception from instruction pointer (LastExceptionFromIP)

When ever a control transfer is done, it is recorded to these registers when enabled.

### 3.1.1 Breakpoint registers

The breakpoint registers is a crucial part of the hardware facilities for debugging, so these will be covered more in depth.

In AMD64, there exists 4 debug address registers (DR0 to DR3) which means there can be 4 breakpoints at a time. The addresses are virtual addresses, except for I/O breakpoints which are I/O port numbers. These virtual addresses are not checked for canonical form [Adv, 2007].

To setup these breakpoints with regard to type and options, debug register DR7 must be setup properly. Each debug address register has a corresponding set of fields in DR7. These settings deal with:
### 3.1 Hardware support for debugging

<table>
<thead>
<tr>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BX</td>
<td>Breakpoint X was encountered. X is 0 - 3</td>
</tr>
<tr>
<td>BS</td>
<td>Single step was encountered</td>
</tr>
</tbody>
</table>

Table 3.1: DR6 debug status register

- **Scope** This can be local or global. Local breakpoints are only present when executing the current task i.e. disabled when switching task. Global breakpoints reside until removed or disabled by software.

- **Type** This can be instruction execution, data read, data read/write or I/O read/write.

- **Length** For data and I/O access, an address range can be specified that will trigger a debug exception. This is on byte, word or double word range from the given address.

The different types of breakpoints have different execution properties. Instruction breakpoints are triggered before the instruction is executed, and data and I/O breakpoints are triggered after instruction execution. This complicates mixed breakpoint types which are located just after each other. But this is a well defined situation, as the different breakpointing methods have given priorities, which are handled by a debug event stack. Instruction breakpoints have lower priority than I/O and data breakpoints [Adv, 2007].

When an address matches a debug register address such as an I/O port, a data access, an instruction that has been executed or a single step, a debug exception (#DB) is triggered. To react upon this, debug register DR6 should be investigated along with the address of the breakpoint located in DR0 - DR3.

A field for each debug address register is present along with a field for single step. This is illustrated in table 3.1. If debug register DR0 was triggered, B0 will be set to 1 by the hardware, and DR1 - DR3 will be set in the same way. B0 to B3 must be cleared by software when the debug handler is done.

The security is ensured by restricting all other tasks than the kernel (running in Current Privilege Level 0) from altering the debug registers. This ensures that user processes do not install breakpoints which may trigger in other processes or in the kernel as this may lead to unexpected behavior.
3.1.2 User space breakpoints vs Kernel breakpoints

Installing breakpoints in user space processes are different from installing breakpoints in the kernel.

In FenixOS, the breakpoint registers are a part of the thread context. Therefore, installing breakpoints into a thread must be done in its context. When a thread is scheduled to run, it will overwrite any existing breakpoints in the debug registers. Therefore kernel breakpoints must be reinstalled when the kernel is invoked.

Furthermore, in a multicore environment each core must have synchronized breakpoint registers or else unpredictable results may occur.

3.1.3 Summary

AMD64 offers a range of debugging primitives which makes it possible to construct a high level debug interface. It has been chosen to use hard breakpoints to provide this interface, instead of soft breakpoints. This eases breakpoint installation and removal. This is because soft breakpoints would require modifications of the virtual memory system regarding Copy-On-Write, page protection, and a lot more. Hardware breakpoints are limited to 4 simultaneous breakpoints and watchpoints, but this has been chosen over the complexity soft breakpoints introduce. The new debug facilities added to AMD64 regarding control transfer single stepping and extended recording facilities has been chosen not to be used. It will later be shown, that the remote debugger does not take advantage of these functionalities and leave them unused at this point in time.
3.2 Remote debugging

3.2.1 Debugger

The term debugger has previously been used multiple times and was briefly introduced in the theory chapter. A debugger is a tool which controls and investigates the execution of a debugged program, or it can perform forensics on a crashed program. These tools can often provide a source code level overview of the progress and the state of the program. An example session is illustrated in figure 3.4.

![Example debugging session](image)

Figure 3.4: Example debugging session

This can not only be done from the executable binary file. Compilers transform source code into a binary file which a machine can execute. This is a complicated process of multiple iterations of transformations and optimizations, which lose more and more details about the original code. Functions may be inlined, instructions may be swapped, etc. Inlined functions are inserted directly into the caller to avoid overhead of stack setup and jumps. It requires some sort of additional information to convert the current instruction pointer into the line of code it originates from - this is handled by adding debugging information to the binary executable file.

There exists many different kinds of formats for adding debugging information [Eager, 2007], but one of the most common is DWARF and it is widely supported. It defines a stan-
standard for the representation of source code and the connection of the source code to the final executable file.

The debugging information is stored as tables inside the binary file. This information includes translation from program counter to source file and line, variable types and data structure information. This can be used be a debugger to give a source level view of the current state and examine variables and data structures.

3.2.1.1 Security

A debugger which is debugging a target process, will have full access to its memory, registers and other state related information. This is a considerable security issue, as private data such as passwords and keys may be read or manipulated by a debugger. This is usually considered harmless in a debugging environment, as the debugger is meant to manipulate and control the debugged process. However, if a running process is involuntarily attached by a debugger, problems may arise. The extent to which a debugger can control a process or a debugged operating system can be limited. Using hard breakpoints and disabling the ability to alter variables and data structures would make it possible to ignore memory write commands from a debugger. Ignoring register writes only disables the ability to manually modify those. This is an uncommon operation. Additional security limitations could be made. Not all registers are actually used when stepping through the program. Registers which are primarily used are the program counter and the stack pointer to keep track of progress. Other registers could be masked and only show the most necessary information to the debugger. Limitations could also be made for memory reads, such that only specific memory ranges were allowed to be read or to restrict memory regions where confidential data such as password hashes and encryption keys may reside.

An example of a debugger is the open source GNU Project Debugger (GDB), which is a full featured program debugger. GDB is compatible with many platforms including AMD64. Many debugging tools use GDB as a backend; an example is the DataDisplayDebugger (DDD) tool which gives a graphical overview of the program and the data structures in the program.

Similar to GDB is the Intel Debugger (IDB). This debugger is closed source but provides an interface similar to GDB.
3.2 Remote debugging

3.2.2 Remote debugging

Remote debugging is a debugging method, where debugging is separated into a target host and a debugger host. Without too much interference, the debugger can control and observe the target host state as it executes. This can be done by adding a small piece of code inside the kernel to which the debugger connects. This can be done in various ways such as using a serial port, a high speed firewire connection or even over an Ethernet network. Remote debugging is often necessary when debugging operating systems. To avoid too much influence on the running code, a remote debugger hooks into code inside the kernel, called a *debugging stub*. This stub is called when a debug exception happens, i.e. a breakpoint or single stepping is triggered, and awaits commands from the remote debugger. This is illustrated in figure 3.5. GDB defines a protocol for remote debugging over serial lines, pipes and networks. This protocol is also supported by IDB. An example of such a remote debugger which uses this protocol is Kernel GDB (KGDB). This debugger infrastructure is incorporated into many UNIX-like systems and provides a kernel debugger stub along with a modified GDB.

![Remote debugger overview diagram](image)

*Figure 3.5: Remote debugger overview*
3.2.3 Communication

When remote debugging, there must be a way to communicate between the debugger host and the target host. This is usually done through either a local pipe when debugging a emulated or virtual machine, a serial line, Firewire or over a network.

RS232 is often used when referring to serial ports in computers. RS232 is a mature standard which has been relied upon since the 1960’s which defines voltage levels, timing and control over a single ended communication line. Even though serial communication using this standard is slow and simple, compared to modern technologies like USB and Firewire, it is very mature, robust and easy to implement. When using a local pipe on a emulated or virtual machine, it can be simulated as a serial port on the target.

Firewire is an advanced high speed serial bus interface standard which offers high speed access between multiple nodes such as computers, cameras, audio equipment, etc. Direct Memory Access is used to avoid the overhead of interrupting the operating system to get attention and the ability of the operating system to move data into memory. This means the firewire controller operates directly with the memory of the attached device. This can be exploited for debugging purposes, as a device may read and alter memory in a connected computer with very little performance loss of the target system [Harvey and Stall, 1991]. Unfortunately, this is also a security issue as devices on the firewire bus may access confidential data on the attached hosts.

A serial port driver can be written with a small footprint, because of the simplicity and configuration of the port driver. This is a considerable strength, as the remote debugging stub cannot debug itself. Both Firewire and a network solution require a very large software stack, however, it can perform at higher speeds and has greater number of functions. The serial COM port has been chosen for its simplicity and robustness.

3.2.4 GDB protocol

GDB provides a protocol for debugging a remote target over a serial communications line.

A debugger sends commands to the stub, which sends responses back to the debugger. This is done in packets which consist of:

- The initial character, which is $
3.2 Remote debugging

- The command or response data
- The data string, which is terminated with a #
- The a two digit checksum, which is calculated as the sum of the data modulo 256. This is illustrated in equation 3.1.

\[
\text{checksum} = \sum_{i=0}^{n} \text{data}_i \mod 256
\]  

(3.1)

When a command has been sent to the target and it has been correctly received, the acknowledge character '+' should be sent back to the debugger. If the packet contained errors which can be detected by checksum mismatch, then the retransmit character '-' should be sent.

A command consists of one or more leading characters, which determine the type of command. This is followed by a list of parameters, which are usually split by ',', or in some special cases by ';'. For example:

Listing 3.1: Example GDB packet

m ffffffff a 0 c 2 1 0 0 0 , 1

Which should be interpreted as command 'm' with parameter 1 as ffffffff a 0 c 2 1 0 0 0 and parameter 2 as 1.

Only a subset of the available protocol are needed for a debugging stub [Fre, 2010]. These commands include:

- Installation and removal of breakpoints (z0 / Z0)
- Read and write registers (g / G)
- Read and write single register (p / P)
- Read and write memory (m / M)
- Single stepping (s)
- Continuing from breakpoint or single stepping (c)
- Respond to general queries from the debugger (Thread id, capabilities etc.) (q)
**Character** | **Command**         | **Implement**
--- | --- | ---
\(z_0\) | Breakpoint installation | Implement
\(z_0\) | Breakpoint removal | Implement
g | Read registers | Implement
p | Read register | Implement
m | Read memory | Implement
s | Single step | Implement
c | Continue execution | Implement
q | General query | Implement necessary queries

Table 3.2: Protocol command subset for a more secure remote debugging stub

As discussed earlier, a debugging infrastructure can be vulnerable to attackers or to uncautious developers. It is possible to use a even smaller subset of the entire protocol than the one listed above, if the extent for debugging is limited. In table 3.2 one can see what needs to be supported by the debugging stub.

**Runtime encoding**

The GDB protocol provides a simple compression algorithm to minimize the data needed to be transmitted. This is called runtime encoding. Identical characters can be collected if they are repeated after each other.

For example, the following data string \(\text{ffffffff802000e8}\) is transformed into \(\text{f}*802000e8\). When the same character is repeated more than 3 times, data could be minimized by encoding. This is done by writing the character that should be repeated, in this case \(f\) followed by "\(*\)". The following character is the number of characters \(n + 29\), which makes it a printable character. "\(*\)" has the decimal value of 36. \(n = 36 - 29 = 7\). This means that \(f\) is followed by 7 additional \(f\)'s.

**3.2.5 Summary**

Remote debugging has been covered in general. Some debuggers have been investigated and the most feasible remote debugger found was the GNU Project Debugger (GDB). It defines a debugging protocol, which has been described. To avoid too high level of complexity and to increase security, only a subset of the GDB protocol should be implemented. The remote debugger will communicate with a kernel stub through a serial line, as this is the most simple and robust communication method known.
3.3 Debugging interfaces for user level processes

As with any other software, software running in user space will contain bugs. The complexity of software increases and this makes it even more difficult to track and correct these bugs. Modern user space software will most likely work with parallelism in threads which introduces a new range of potential bugs, including synchronization problems. This includes race conditions, deadlocks, livelocks and etc. Most likely the software will also interact with what surrounds it, such as other software and the kernel. This increases complexity which increase the potential number of defects.

For this reason, a modern operating system must provide some debugging primitives for making it feasible to develop software running on it. There exists many methods and primitives which vary in different operating systems, however they can generally be categorized into the following:

- Tracing / Observing
- Controlling / Examining
- Postmortem Analysis / Investigating

As mentioned earlier, tracing primitives can consist of bare print statements, however, they can also be provided by the kernel. This is usually in reference to a system call, which is the method for a user process to communicate with the kernel. The kernel may track each system call with parameter values for a given user process, and this may give a developer a good way of locating the problem.

Controlling software is usually done with a debugger. A user space debugger differs from the previous description of a debugger in that this kind of debugger does not have the privilege to exploit the hardware support for debugging by itself. Neither does it have permissions to alter other processes, as this is the core concept of process isolation. An operating system must provide an interface for this special kind of processes to:

- Install breakpoints in a process. This would require either write access to the text segment of the debugged process or access to write to its debug registers.
- Enable single stepping. This would require access to rFlags register of the debugged process.
- Read registers of the debugged process. This is needed to locate the instruction and stack pointer which is used to determine the current position in the source code.
- Read and write to memory of the debugged process. This is needed to get stack information and to display and alter variables and data structures.

Furthermore, an infrastructure for invoking the debugger when breakpoints are encountered in the debugged process must be provided. An illustration of a debugging session with such an infrastructure is illustrated in figure 3.6. An example of such an infrastructure is ptrace, which can be found in many UNIX-like systems. This interface defines ways to attach a debugger to a process, single step, writing to memory, continue and a lot more [BSD, 1994].

![Figure 3.6: User process debugging](image)

Precautions should be taken, when providing these debugging features to processes. When a process is being debugged, all memory can be read by the debugger, which makes it vulnerable for tampering of private data. Proprietary software may be reverse engineered, etc. For this reason, some policy should exist to only allow processes in a debugging scenario to be debugged. The traditional policy is to only allow a debugger to hook into a child process. This means that the debugged process must be spawned by the debugger. This is not always possible, if running servers or other long lasting processes should be debugged. For this reason, a process can often mark it self as debuggable. Letting a debugger hook into the running process.

Newer infrastructures may use digital signature schemes to sign processes, such that only debuggers which have been signed with the same certificate as the process may
attach to it.

As previously mentioned, breakpoints can be implemented by using the hardware debug registers. A simple infrastructure would be to provide an thin layer in the kernel which would allow manipulation of the debug registers of another process. When a breakpoint was encountered in a process, it would be put to sleep and the debug process which installed the breakpoint could be awoken. This scheme follows the one illustrated in figure 3.6. It is a simple scheme which leaves the logic of breakpointing to the debugger and leaves a minimal footprint into the kernel.

A consequence of the traditional scheme for debugging is a high number of context switches are needed. Single stepping a program requires descheduling of the debugged process, rescheduling of the debugger which after a short while would be descheduled and the debugged process rescheduled again. This is a major performance issue and could have a major influence on debugging of concurrent programs. Problems which may only trigger in special timing scenarios, for example. Newer infrastructures try to limit the amount of context switches needed to debug a running process. Some may even try to eliminate the need for context switching [IBM, 2007].

Postmortem analysis of user space programs is much like postmortem analysis of a kernel. This will be covered in the next section. It is worth noting that user space debugging primitives could be left out and only provide a core dump. This dump can be used to create a backtrack of what happened.

### 3.3.1 Summary

It has been chosen to provide a thin layer between a debugger process and the debugged process' debug registers. This has been chosen instead of providing a breakpoint abstraction inside the kernel. This has been done because it limits the complexity introduced to the kernel and leaves the breakpoint logic to the user space processes. As stated in section 3.1.3, soft breakpoints introduces a considerable level of complexity to the virtual memory system. Therefore, only hardware breakpoints should be implemented.
3.4 Postmortem analysis

When an operating system fails, it often leaves the system in an unusable state. Sometimes even nonresponsive for keyboard inputs. To make it possible to investigate the reason for the system crash and perform postmortem analysis, a core dump can be created when the system fails. This is an mature technique from that time where batch processing was used and computer time slots had to be scheduled. When a program failed a print out of the memory was made and off-line analysis of the failure could be done. Postmortem analysis has multiple strengths:

- Comparison between failures can be done. The frequency of a defect may be recorded or patterns in the core dumps can be found, which may help to locate the problem.

- Analysis and debugging can be done offline. This can be a strength in embedded systems, which may not have any screen or input devices. Core dumps can even be sent to the manufacture, so defects can be removed in later revisions of the software.

- Minimal influence on debugged software. Postmortem analysis has no or minimal influence compared to attaching a debugged into a process. The program will run as normal until it crashes, where the core dump could be produced.

A core dump is a dump of the contents of the memory along with other state information, such as the cpu registers. This is saved to some non-volatile storage. A library like BFD can simulate memory and register access from a core dump [Fre, 2009]. This is illustrated in figure 3.7. To perform postmortem analysis on source level, debugging information must be provided as earlier described. Low level analysis can still be performed, however the debugger can provide more information to the developer, if it can interpret the dump file.

There exists multiple formats for these core dumps, but the most common in UNIX-like systems is the Executable and Linkable Format (ELF).

When an operating system fails, it is often necessary to tell the user and the developer what went wrong. This is often called a kernel panic. The purpose of this panic is to provide enough information which gives an idea of what and where the problem resides. This, along with a core dump is commonly produced when the system fails, and give developers primitives to locate and fix a potential bug.
3.4 Postmortem analysis

Figure 3.7: Example session of a core dump analysis

3.4.1 Summary

A very useful method for doing postmortem analysis is core dumping. Unfortunately, FenixOS does not provide filesystem support yet and this complicates core dumping tremendously. Instead, a new kernel panic facility could be provided to display information related to the state of the system when it crashed.
The design chapter will purpose a design based on the choices made in the analysis. The resulting design should be at such a level that it is possible to implement a prototype.

### 4.1 Problem statement

A debugging infrastructure is going to be designed and a prototype will be implemented. The usefulness and future improvements for both are going to be evaluated and discussed.

This debugging infrastructure will contain:

- A remote debugging stub which is debuggable with GDB over a serial line. For simplicity, only a single core system is supported. For security reasons, only a subset of the GDB protocol is supported which provides the functionality necessary to single step, breakpoint, investigate data structures and backtrack execution.

- An interface for user space debugging providing the primitives needed for installing and removing breakpoints in a debugged process. This is done through
a thin layer in the kernel, which provides access for a debugger process to the debug registers of a debugged process. When a debugged process encounters a breakpoint, the debugger should be notified.

- Functionality similar to Panic, which is called upon kernel crash. This should provide information such as register values and error codes, and explain the exception number to track the cause of the defect.

### 4.2 Design proposals

To motivate a design, multiple solutions to the stated problem must be considered.

A central issue for such a solution for a hybrid kernel, is whether to implement a debugging infrastructure as a user space server or integrate it into the kernel. Both solutions are plausible. Illustrations of the organizations are depicted in figure 4.1a and 4.1b.

However, when implementing a debugger, breakpoints will stop execution of what was ongoing on the present time regardless if a breakpoint is encountered in supervisor mode or user mode. FenixOS does not and is not supposed to implement callable user space servers, which complicates this tremendously. If the kernel encounters a breakpoint, it must schedule processes until the debugging process is scheduled.
4.3 Debug infrastructure design

Figure 4.2: Design options for kernel debugging interfaces

and can take action. The kernel may have changed its state since it encountered the breakpoint and leave data examination untrustworthy.

Another issue is, what level of abstraction that should be provided regarding debugging facilities inside the kernel. A unified interface to the debugging facilities could be one solution. This could provide a generic interface for creating high level debug facilities. Another alternative is to avoid this high level of abstraction and the complexity that this kind of infrastructure would require. This approach can expose debugging facilities directly to for example the CPU and Thread class for breakpoint installation and removal. Illustrations of these are depicted in 4.2a and 4.2b.

If a wider span of debugging facilities should be provided by the kernel, a unified debug interface would be a good solution. However, in this thesis only remote debugging and user space debugging should be provided. This makes the unified debug interface a complex solution for this problem and adds unnecessary overhead.

The same issue arises, when designing the interface for debugging user space processes. A unified high level interface could be designed. This interface would have to define an abstraction for breakpoints, watchpoints, single stepping, etc. Another approach, is to provide a low abstraction and leave debugging logic to a user space debugger. The two solutions are illustrated in figure 4.3a and 4.3b.

4.3 Debug infrastructure design

The debugging infrastructure will be a part of the kernel i.e. not running as a user space server. This is chosen for simplicity and to avoid the overhead of additional context switches when debugging user space processes.
Remote debugging and user process debugging are separated into a Remote debugging class and the system call interface respectively. The design of the remote and user process debugging is illustrated in figure 4.4.

- Both interfaces can install breakpoints. The remote debugger can install and remove breakpoints per CPU basis and the user process debugging can install and remove breakpoints per thread basis.
- When a breakpoint exception is triggered in kernel mode the remote debugger is called. If the exception is triggered in user mode the debugger debugging the current thread is called.

### 4.3.1 Remote debugger

To separate the serial port communication and GDB protocol parsing from the debugger logic, the organization is as illustrated in figure 4.5.

- A generic serial port driver takes care of port initialization, reading and writing to the port. This can be done on a per string basis.
- The GDB protocol parser parses a GDB packet into a `GDBcommand` class. This class contains the command type along with the associated parameters. The parser also encodes packets which are going to be sent as response to the debugger.
4.3 Debug infrastructure design

Figure 4.4: Illustration of the workings of the remote debugger and user space debugging

Figure 4.5: Organization of the remote debugger
4.3.2 Postmortem analysis

As argued in the analysis, a core dumping facility is not going to be a part of the debug infrastructure at this point in time because of the lack of a non-volatile storage in FenixOS. But a facility for reporting and displaying useful information regarding the conditions present while the system crashed is going to be designed and implemented. This kernel panic handler is called `oops`. The design for this is illustrated in figure 4.6.

![Diagram](image)

- When an unhandled exception happens in supervisor mode, the registers of the current CPU are displayed on the screen. When this is done, the CPU halts to prevent further damage of the system state.

- When an unhandled exception happens in user mode, the current threads registers are displayed on the screen. Execution continues, as the rest of the system should be in a sane state.

In both cases, a description should be found for the current exception number and it should display the error code, if present.
4.4 Project management

This section describes the methods that has been used regarding project management to complete this thesis.

4.4.1 Time plan

The project planning has to some extent made use of the Agile Unified Process (AUP) [Ambler, 2009]. This involves a split of the development process into 4 phases which is Inception, Elaboration, Construction and Transition. These phases can be found in the time plan in table 4.2.

In the inception phase, the project and risks should be defined. On this basis, the feasibility of the project should be determined. The risks for this project are discussed in section 4.4.2.

In the elaboration phase, the architecture of the system should be created and validated and sufficient plans for entering the construction phase should be made.

In the construction phase, iterations of coding and testing is done. In this thesis, the set of iterations is listed in table 4.1.

In the transition phase, the product is being finished. This involves bug fixes, validation of correctness and deployment preparation.

The entire time plan is listed in table 4.2.

<table>
<thead>
<tr>
<th>Iteration #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Be able to install breakpoints</td>
</tr>
<tr>
<td>2</td>
<td>Be able to transfer control to correct handler on breakpoint exception</td>
</tr>
<tr>
<td>3</td>
<td>Be able to read memory from kernel and selected threads</td>
</tr>
<tr>
<td>4</td>
<td>Be able to pass a minimal set of the GDB protocol - test with actual GDB debugger</td>
</tr>
<tr>
<td>5</td>
<td>Provide debug interface to user space processes</td>
</tr>
<tr>
<td>6</td>
<td>Get and display relevant data in case of crash</td>
</tr>
<tr>
<td>7</td>
<td>Be able to evaluate and display test cases</td>
</tr>
</tbody>
</table>

Table 4.1: Project iterations
<table>
<thead>
<tr>
<th>Phase</th>
<th>Week no.</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception</td>
<td>5</td>
<td>Lifecycle Objectives (LCO)</td>
</tr>
<tr>
<td>Elaboration</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Construction</td>
<td>7</td>
<td>Lifecycle Architecture (LCA)</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Iteration #1</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>Iteration #2</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>Iteration #3</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
<tr>
<td></td>
<td>13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>14</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>Iteration #4</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>Iteration #5</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>Iteration #6</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>Iteration #7, Initial Operational Capability (IOC)</td>
</tr>
<tr>
<td>Transition</td>
<td>21</td>
<td>Product Release (PR)</td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>Report writing</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>Report writing</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>Report refactoring</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>Report refactoring, Deliver project</td>
</tr>
</tbody>
</table>

Table 4.2: Project time plan
4.4 Project management

4.4.2 Risk analysis

Risks are rated with a severity level between 0 and 5. 5 is high-severe and 0 is non-severe.

---

**#1 Breakpoints can not be installed**

<table>
<thead>
<tr>
<th>Severity: 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>There may be differences in implementations of the debug registers from machine to machine. This should be unlikely, because the basic functionality in the debug registers dates from x86 architecture and thus backward compatible and available in long, compatibility and protected mode. It is crucial for the project that one of the breakpoints work</td>
</tr>
</tbody>
</table>

---

**#2 BreakpointHandler can not determine breakpoint reason**

<table>
<thead>
<tr>
<th>Severity: 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the breakpoint handler can not determine the reason for the breakpoint which had just occurred, it would be very hard to recover from that breakpoint, if not impossible. If it is not possible to determine the reason of breakpoints, only one type of breakpoints should be used, to recover from that one.</td>
</tr>
</tbody>
</table>

---

**#3 BreakpointHandler can not recover from breakpoint**

<table>
<thead>
<tr>
<th>Severity: 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the breakpoint handler can not recover, i.e. it can not restart interrupted instruction or bring instruction to original state, it is not possible to provide instruction break points. In that case, only data break points can be used. This is not supported by GDB, and remote debugging will not be possible. The ability of the breakpoint handler to recover from a breakpoint is crucial for the project!</td>
</tr>
</tbody>
</table>

---

**#4 RDebug can not read kernel memory**

<table>
<thead>
<tr>
<th>Severity: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the remote debugger can not read from kernel memory, the information provided by the remote debugger becomes very limited, as the state of kernel can not be inspected during execution. It should still be possible to install breakpoints and single step and inspect the flow through the kernel.</td>
</tr>
<tr>
<td>#5</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td><strong>Severity:</strong></td>
</tr>
<tr>
<td>If the <strong>GDBParser</strong> can not communicate with GDB, remote debugging with GDB will not be possible. A custom proof of concept remote debugger can be written for testing purposes, if the <strong>GDBParser</strong> fails to communicate with GDB.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#6</th>
<th><strong>User space debugging can not read thread memory</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Severity:</strong></td>
<td>3</td>
</tr>
<tr>
<td>If the User space debugging interface can not read from thread memory, thread debuggers get very limited, as the state of the thread can not be inspected during execution. It should still be possible to install breakpoints and single step the thread and inspect the program flow.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#7</th>
<th><strong>User space debugging can not communicate with user space processes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Severity:</strong></td>
<td>5</td>
</tr>
<tr>
<td>If user space debugging can not communicate with user space processes, it is not possible to support thread debuggers. This is unlikely as a syscall interface is already present in the FenixOS kernel.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#8</th>
<th><strong>User space debugging can not install breakpoints</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Severity:</strong></td>
<td>5</td>
</tr>
<tr>
<td>If user space debugging can not install breakpoints, it is not possible to support user space debuggers. This may be a issue, as the remote debugger only runs and operates in kernel space and therefore does not test the ability of the debugger core to break point user space threads.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#9</th>
<th><strong>Tester can not evaluate given test cases</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Severity:</strong></td>
<td>1</td>
</tr>
<tr>
<td>If the testing framework does not work, it is not possible to provide automatic functional testing. In this case manual testing should be used instead to document the functionality and stability of the system. In either case, the infrastructure design will be discussed and solutions to debugging issues will be provided.</td>
<td></td>
</tr>
</tbody>
</table>
4.4.3 Project progress

The project time plan has evolved during the project period. Some iterations showed to be less complex than expected and other iterations showed to be quite complex. The latter has especially been the case for iteration #3 which was the implementation of the GDB protocol. The original architecture has also changed during the project. This has also had effect on the iterations, as these was created with the design as basis. The time plan and iterations shown in table 4.2 and 4.1 are the final versions.

AUP incorporates technical documentation during the development process. This has not been used to an sufficient extent and could have eased the process of documenting this report.

In general, this thesis has changed its design or architecture during the project. This reflects that not enough time has been used to thoroughly investigate the possibilities available before designing and implementing. The documentation phase should have been incorporated in the entire process but has been postponed to the last. The construction phase grew as complexities and problems was found, which cut time for documentation and finishing the project. This is an unfortunate situation.
This chapter will cover the technical details for the implementation of the debug infrastructure designed in the previous chapter.

5.1 Remote debugging

The classes which relate to remote debugging are listed in table 5.1.

The remote debugger works as follows:

- When a breakpoint is encountered, a debug exception (#DB) will be triggered. This will call the interrupt handler.

- The interrupt handler looks at the exception number and determines that it is #DB. The BreakpointHandler is invoked with handleKernelBreakpoint().

- The BreakpointHandler will investigate the cause of the exception by looking at debug status register DR6. If a breakpoint was encountered, one of B0 - B3 will be set as 1. If a single step triggered the exception then BS will be 1. If none of them is set - the debug exception was triggered manually with the INT
<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BreakpointHandler</td>
<td>Is invoked by the interrupt handler when a breakpoint exception (#DB) occurs.</td>
</tr>
<tr>
<td>GDBParser</td>
<td>Parses GDB packets into a GDBCommand class. Is also responsible for encoding data into GDB packets.</td>
</tr>
<tr>
<td>UARTDriver</td>
<td>Initializes, reads and writes strings to a given serial port.</td>
</tr>
<tr>
<td>CPU</td>
<td>Existing abstraction of a CPU. Methods have been added to install and remove breakpoints into a given CPU.</td>
</tr>
<tr>
<td>RDebug</td>
<td>Uses GDBParser and UARTDriver to poll for new commands from the connected debugger. It takes action upon the received packets, installs and removes breakpoints, reads memory areas, etc.</td>
</tr>
<tr>
<td>StringHelper</td>
<td>Is used to contain various of helper functions, such as hex to integer conversions, etc.</td>
</tr>
</tbody>
</table>

Table 5.1: Classes for remote debugging

![Diagram](image_url)  
Figure 5.1: Class relations for remote debugging
5.1 Remote debugging

<table>
<thead>
<tr>
<th>Handler</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stepHandler</td>
<td>Handles single stepping packets &quot;s&quot;</td>
</tr>
<tr>
<td>generalQueryHandler</td>
<td>Handles query packets &quot;q&quot;</td>
</tr>
<tr>
<td>lastSignalHandler</td>
<td>Handles last signal packets &quot;?&quot;</td>
</tr>
<tr>
<td>readRegistersHandler</td>
<td>Handles register reads of all common registers packets &quot;g&quot;</td>
</tr>
<tr>
<td>readMemoryHandler</td>
<td>Handles memory reads packets &quot;m&quot;</td>
</tr>
<tr>
<td>readRegisterHandler</td>
<td>Handles register reads of a specified register &quot;p&quot;</td>
</tr>
<tr>
<td>insertBreakpointHandler</td>
<td>Handles breakpoint installation packets &quot;z0&quot;</td>
</tr>
<tr>
<td>removeBreakpointHandler</td>
<td>Handles breakpoint removal packets &quot;Z0&quot;</td>
</tr>
<tr>
<td>emptyHandler</td>
<td>A common handler for ignoring the command and responding with an empty packet.</td>
</tr>
<tr>
<td>OKHandler</td>
<td>A common handler for ignoring the command and responding with a OK packet.</td>
</tr>
</tbody>
</table>

Table 5.2: Packet handlers

$3$ command. This command forces an interrupt with a given number to be triggered. In either case, if the cause could be determined then call RDebug’s handleSignal() or else call handle().

- When handleSignal() is called, it starts by sending a signal packet in response to the attached debugger. For both handle() and handleSignal() the processLoop() will be called.

- processLoop() reads data from UARTDriver via read() and is parsed by GDBParser via parse(). This results in a GDBCommand object with a command type and parameter values.

- processInput() is called to take action upon this GDBCommand. Here a proper handler is called to handle the given packet. These handlers are listed in table 5.2.

The class relation mentioned is illustrated in figure 5.1

5.1.1 Initialization

The initialization sequence is not documented very well, and has been reversed engineered packet-by-packet extending the handlers to provide the necessary information.
to the debugger.

An example of an initialization sequence is listed in listing 5.1. <-- indicates received command and --> indicates sent response. The packets are described in the table present in table 5.3. This is an example of a GDB 7.1 initialization sequence, however it may differ between versions. Handlers for each type of packet have been implemented. If the response required is only OK or an empty packet, a common OKHandler() and EmptyHandler() are used respectively.

Listing 5.1: Example GDB initialization sequence

```
<-- $qSupported #37
--> $#00
<-- $Hg0#df
--> $OK #9a
<-- $? #3f
--> $S05 #b8
<-- $He-1 #09
--> $OK #9a
<-- $qC #b4
--> $OK #9a
<-- $qAttached #8f
--> $OK #9a
<-- $qOffsets #4b
--> $Text =0; Data =0; Bss =0 #04
<-- $g #67
--> $00000000000000056182280ffffff0200000000000019
000000000000000000000000000080ffffff19000000
00ffffff00000000000000000b5442080ffffff46020001800000
20000000000000000000000000000000000000000000 #16
<-- $mfffffff802015d6 ,1 #c4
--> $e #9d
<-- $mfffffff802015d6 ,1 #c4
--> $e #9d
<-- $qSymbol:: #5b
--> $OK #9a
<-- $Z0 , ffffffff802000e8 ,1 #0a
--> $OK #9a
<-- $vCont? #49
--> $#00
<-- $Hc0 #db
--> $OK #9a
```
### 5.1 Remote debugging

<table>
<thead>
<tr>
<th>Packet</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qSupported</td>
<td>Query packet which asks for debugging stub features. This is responded to with an empty packet which indicates no additional features are available.</td>
</tr>
<tr>
<td>Hg0</td>
<td>Sets the current thread id to 0 for general operations. This is ignored as kernel threads are not available.</td>
</tr>
<tr>
<td>?</td>
<td>Asks how the debugger reached the present state. This is responded to with a signal number. This signal number uses the POSIX signal scheme, where signal 5 is the breakpoint signal.</td>
</tr>
<tr>
<td>Hc-1</td>
<td>Sets the current thread id for step and continue operations. This is also ignored.</td>
</tr>
<tr>
<td>qC</td>
<td>Asks for the current thread id. If any reply other than an integer is returned, it is ignored.</td>
</tr>
<tr>
<td>qAttached</td>
<td>Asks if debugging stub is attached.</td>
</tr>
<tr>
<td>qOffsets</td>
<td>Asks for relocation offsets. This will be relevant if the kernel had been relocated to another address than 0, but this is not the case for FenixOS.</td>
</tr>
<tr>
<td>g</td>
<td>Asks for register values for all common registers</td>
</tr>
<tr>
<td>mffffffff802015d6,1</td>
<td>Asks for 1 byte of data starting at address 0xffffffff802015d6. This is the address pointed to by the instruction pointer RIP.</td>
</tr>
<tr>
<td>qSymbol::</td>
<td>Indicates the debugger is ready for symbol look ups from the debugging stub.</td>
</tr>
<tr>
<td>Z0,ffffffff802000e8,1</td>
<td>Removes a breakpoint at address 0xffffffff802000e8. This is done because the debugger thinks that it encountered a breakpoint.</td>
</tr>
<tr>
<td>vCont?</td>
<td>Asks if extended continue operation is available. This is not the case and is just responded to with an empty packet.</td>
</tr>
<tr>
<td>Hc0</td>
<td>Sets the current thread id for step and continue operations. This is ignored.</td>
</tr>
</tbody>
</table>

Table 5.3: GDB Initialization
5.1.2 Breakpointing

When installing and removing breakpoints, the debugger issues "z0" and "Z0" packets respectively. When continuing execution and encountering a breakpoint, the stub must inform the debugger. When continuing, the debugger stalls and awaits a signal packet from the stub. This is done by \texttt{handleSignal()}.

5.1.3 Single stepping

When enabling single stepping, the trap flag (TF) if the current processors \texttt{rflags} register is set to 1. When the interrupt handler returns, single stepping will be enabled and it will trigger a debug exception after execution of the next instruction. Single stepping behaves like breakpointing, \texttt{handleSignal()} is called and signals to the debugger execution has stopped. Single stepping is disabled when entering the interrupt handler by hardware, and the trap flag (TF) is also cleared by the debugging stub.

5.1.4 Variable and data structure examination

Variable and data structure examination is unknown to the debugging stub which only provides the necessary memory contents from specified memory addresses. The interpretation is done by the debugger, based on the debugging information present.

Notes about protocol byte order

During development of remote debugging, it was noticed that the GDB interprets transferred data from register and memory reads in big endian. GDB was setup to use little endian, however this did not change the byte order needed when transferring data. No documentation related to this anomaly has been found. Byte order helper functions has been implemented to deal with this issue.

5.2 User space debugging

The user space debugging interfaces have been implemented as a thin layer between a debugger and the debug registers of the debugged process. This moves all breakpointing logic to the debugger for breakpoint setup (DR7), address installation (DR0-DR3) and breakpoint status (DR6).
An example of the syscalls of a debugger in a debugger session is illustrated in figure 5.2.

The syscalls made are described in table 5.4.

The scheduler that FenixOS provides was not functional when this user space debugging was developed. For this reason, a very rough, unfair and ineffective round-robin scheduler was implemented for testing purposes. A new method has been added to the `ReadyQueue` class, called `rrSchedule()`. This scheduling method does the following:

- If a previous thread is present, add it to the end of a thread queue. This data structure is called `Debug::Queue`.
- If the previous thread is not put to sleep, mark it as ready.
- Get the thread at the head of the thread queue. If this thread is sleeping or not active, put it to the end of the thread queue. Continue until a ready thread is present and schedule it for execution.
<table>
<thead>
<tr>
<th>Syscall</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBUG_THREAD_TRACE</td>
<td>To let a debugger process access or modify registers, or anything else, in a debugged process. The debugged process must be known and registered. Security mechanisms for authenticating the debugger process can be implemented in this syscall. At the present state, no additional security mechanisms has been implemented.</td>
</tr>
<tr>
<td>DEBUG_THREAD_READDRX</td>
<td>Reads the DRX register of the registered debugged process. Where X can be 0 to 3</td>
</tr>
<tr>
<td>DEBUG_THREAD_WRITEDRX</td>
<td>Writes to the DRX register of the registered debugged process. Where X can be 0 to 3</td>
</tr>
<tr>
<td>DEBUG_THREAD_READRIP</td>
<td>Read the instruction pointer of the registered debugged process. This is used when a breakpoint is encountered. The debugger can lookup the instruction pointer in the debugging information provided by the debugged process’ executable and get the corresponding source file and line.</td>
</tr>
<tr>
<td>DEBUG_THREAD_CONT</td>
<td>Continues the execution of the registered debugged process, when it has been stopped by a breakpoint.</td>
</tr>
</tbody>
</table>

Table 5.4: Debugging syscalls
5.3 Postmortem analysis

A core dumping facility will not be implemented. However, a facility for reporting useful information when the system crashes will be implemented.

When a interrupt or exception occurs, a handler will be called from the interrupt vector. This interrupt vector is located in `inthandr.s`. A macro `irq_macro` has been made to ensure that most of the interrupts and exceptions are handled the same way, in a sense that the handler code is wanted in C++ instead of assembler language. The macro calls an appropriate handler in `InterruptRoutines.cpp` based on whether the exception happened in user or supervisor mode and if an error code is present.

An interrupt or exception is determined as unhandled, if the interrupt number which was passed to one of the handlers in `InterruptRoutines.cpp` is not known. At this present state, only few interrupts or exceptions are handled. Only debug exceptions are handled, which calls the `BreakpointHandler`. Prior to the `Oops` class which implements new panic methods, a simple print statement was made upon a unhandled exception.

Figure 5.3: Panic messages prior to `Oops`

As mentioned, the class `Oops` has been implemented to take care of displaying panic messages upon unhandled exceptions or interrupts. `kernelPanic()` and `userPanic()` is called when an unhandled interrupt or exception happens in supervisor and user
mode respectively.

The following takes place, when a panic is called.

- The exception number is used as an index into an array of exception descriptions. This description is displayed on the screen. A message can be passed which will be displayed if present.
- All common register values are displayed on the screen.
- The stack pointer RSP and the instruction pointer RIP are fetched from the interrupt stack and displayed on the screen.

The registers and interrupt stack access differ between interrupts and exceptions in user mode and supervisor mode. In user mode, both registers and values from the interrupt stack are available from the current thread object.

When an interrupt or exception occurs, certain values are push onto the stack [Adv, 2007][p244]. This includes the stack pointer (RSP), rFlags, Code segment offset (CS) and the instruction pointer (RIP). These are important registers display in case of failure.

In kernel mode, the registers are fetched from the current CPU and stack individually. Screen dumps of user panics and kernel panics are illustrated in the evaluation chapter.
This chapter will evaluate the implemented solution to the stated problem. This will be done with functional tests of each component individually which should show the correctness of the solution. A small testing infrastructure has been made to test the GDBParser.

Environment

FenixOS is tested and run on a virtual machine. This is done to avoid damage on real hardware, to some extent adding additional debugging facilities and control of hardware setup. The virtual machine used in this setup is Sun VirtualBox 3.1.6 on Mac OS X 10.6.

6.1 Remote debugging

VirtualBox can simulate 2 serial ports. There are multiple options:

- Read and write to pipes
- Write only to a raw file
Evaluation

<table>
<thead>
<tr>
<th>Target</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open_serial</td>
<td>Converts pipe into serial device. The device created will be displayed.</td>
</tr>
<tr>
<td>gdb</td>
<td>Opens GDB with kernel image loaded and architecture set.</td>
</tr>
<tr>
<td>gdbtui</td>
<td>Opens GDB with kernel image loaded and architecture set. This is with a text based user interface</td>
</tr>
</tbody>
</table>

Table 6.1: Debug make targets

- Read and write to device

In this setup, serial port 1 is setup as a pipe for attaching the remote debugger and serial port 2 is setup as a raw file for logging debugging information during execution.

GDB has to be compiled to target a 64 bit architecture. In this setup, GDB was compiled against x86_64-pc-linux-gnu. If another target is used, register order and selection may vary when GDB interprets the "g" command. GDB does not support the use of pipes directly for remote debugging, therefore a tool must be used to convert this pipe into an emulated serial device. The tool used was socat which opens a pipe and creates a serial device to which GDB can connect.

To ease the procedure of opening pseudo terminals and start GDB with the kernel image loaded, additional make file targets have been made. This is illustrated in table 6.1

When the virtual machine has been started, FenixOS will indicate it is ready for remote debugger connection. This is depicted in figure 6.1.

When the virtual machine is ready, the pipe must be converted into a serial device. This is shown in figure 6.2.

When the serial device is setup, GDB should connect. This is illustrated in figure 6.3.

6.1.1 Tests

The following functional tests will be carried out:

- Breakpointing into a function
6.1 Remote debugging

Figure 6.1: FenixOS remote debugging test. Awaits connection from GDB

Figure 6.2: FenixOS remote debugging test. Prepare serial device
Figure 6.3: FenixOS remote debugging test. Attach debugger
6.1 Remote debugging

- Single step through code
- Back track execution
- Examine data structures and variables

Breakpointing

To test breakpointing, functional tests has been made. Figure 6.4 depicts a debug session where a breakpoint is inserted at `FenixOS::System::Kernel::ProcessManagement::ReadyQueue::schedule()`. Execution is continued with the command "continue" or "c". The breakpoint is encountered and the debugger shows the current position. This is shown in figure 6.5.

```
519 Debug::printTestRunTests();
520 Debug::exit();
521 /* create processes for all kernel resident images. */
522 ProcessManagement::createProcesses();
523 #ifdef DEBUG_GDB_ENABLED
524 #define GDB_TRUE 1
525 #else
526 #define GDB_TRUE 0
527 #endif
528 Debug::screen.Write("Please connect GDB client...\r\n");
529 Debug::screen.Write("Starting Process: \r\n");
530 Debug::screen.Write("Starting Process: \r\n");
531 /*
532 * Main:
533 */
534 ProcessManagement::readyQueue.schedule();
535 #ifdef GDB_TRUE
536 // do some stuff
537 #endif
```

Figure 6.4: Functional test of breakpointing: Breakpoint is inserted

Single stepping
Figure 6.5: Functional test of breakpointing: execution continues and breakpoint is encountered
To test single stepping, functional tests has been made. Figure 6.6 and 6.7 shows a simple test of single stepping. In the first figure, execution has reached to line 114 in the `Queue.hpp`. A single step command is issued and execution stops at the line 117. One might wonder, why 3 lines are skipped in one single step. This is because of the compiler optimization level used when compiling the kernel. This may inline functions, instructions might be swapped and this can introduce some confusion while stepping through the code.

![Code snippet from `Queue.hpp`](image)

**Figure 6.6:** Functional test of single stepping: execution has stopped and is currently at line 114

### Back tracking

Backtracking displays a representation of the current call stack. This gives a picture of the execution path that has been taken to get to a certain point. A backtrack command "bt" has been run in figure 6.8.

### Data structure examination

Data structure examination is a very important tool while debugging, as this can give a picture of the state of the program. This can help tracing infections in the state and locate a potential defect. A screen driver object called `Screen` is examined in figure 6.9.

### GDBParser
Figure 6.7: Functional test of single stepping: single step issued with command "s" and now execution has stopped at line 117

A small testing environment has been created to make unit testing-like test cases [Andy Hunt, 2004]. As discussed in the discussion chapter, unit tests can be difficult to apply to operating systems. However, testing of a protocol parser is possible and has been done. The output from the test is depicted in figure 6.10 and 6.11.

6.2 User space debugging

A debugging interface for user space debugging has been implemented. To test this, two user programs has been made: a tracee which is a debugged process and a tracer which is a debugger process. A test scenario has been put up:

- The tracer attaches to the tracee with the DEBUG_THREAD_TRACE syscall. The thread address of the tracee must be supplied as parameter.
- The tracer installs a breakpoint into the tracee with installBreakpoint(). This function manipulates with the debug registers of the tracee. This breakpoint is a data read/write breakpoint, as this is easier to trigger as we do not know
Figure 6.8: Functional testing of back tracking: Inside the screen driver, a back track command "bt" is issued.

Figure 6.9: Functional test of data structure: Examining the Debug::screen object.
Figure 6.10: Unit test of GDBparser: initialization sequence

Figure 6.11: Unit test of GDBparser: operation
6.3 Kernel and user panics

- The tracer waits for tracee to encounter the breakpoint and goes to sleep with `WAIT` syscall.

- Tracee manipulates with the variable which the breakpoint points to. This triggers the breakpoint and the thread goes to sleep.

- The tracer is awoken and continues execution of tracee with `DEBUG.THREAD_CONT` syscall.

This is shown in figure 6.12. Lines starting with '+' are written by the tracer and lines starting with '-' are written by the tracee. The lines should related closely to the described procedure above.

![Screenshot of a computer interface with text](image)

Figure 6.12: Functional test

### 6.3 Kernel and user panics

As with the other components, kernel and user panics are functionally tested. To test these, an unhandled interrupt was issued manually in kernel and user mode respectively. This was done with the `INT $5` assembler instruction. The results is depicted in figure 6.13 and 6.14 respective.
Figure 6.13: Kernel panic: A kernel panic was provoked by issuing an unhandled interrupt.

Figure 6.14: Kernel panic: A user panic was provoked by issuing an unhandled interrupt while running a process.
6.4 Summary

The overall reliability of the implemented remote debugger is acceptable, but is still in an immature state. It is possible to breakpoint, single step, run backtrack and examine data structures. Some “g” packets are determined as too long by GDB and is a sporadic defect. In GDB 7.1, the file `gdb/remote.c` contains remote debugging related code. Line 5396 evaluates the length of the received "g" packet against a architecture specific `rsa->sizeof_g_packet` variable. This variable is set to 100. This complicates register reads, as this is 328 characters of data. Runtime encoding has helped increase the stability, but the fault is triggered sometimes anyway. Means to extend the packet size could be investigated.

Backtracking unwinds too far and provokes a kernel panic caused by page fault.

The user space debugging interface is also very immature and needs to be extended to provide a more complete debugging facility. At this point in time, a debugger process can install and remove breakpoints in another thread, get notified when a breakpoint is encountered and start execution of the debugged thread again. The solution is quite robust but requires knowledge of the thread address to be traced. This can change if other sections of the executable kernel image changes size.

User and kernel panics have been implemented and works in an expected manner. An exception message is displayed on the screen along with the current registers. Unwinding the stack to produce a stack trace is not an easy task and has not been done, as debug tables in the executable file must be parsed. This is because compilers may leave stack creation out when making function calls and other optimizations to increase performance.
Remote debugging

As stated, only a subset of the GDB protocol has been implemented. This limits the extent in which GDB can debug FenixOS. Data structures and variables along with the registers cannot be changed by the debugger. This is chosen by design to avoid too high level of complexity along with making the debugging stub more secure. The remote debugger is written specific for AMD64 with a GDB compiled against a 64 bit Linux architecture. If GDB was compiled against another target, register selection and order may be different and this is not handled at this present state. Usually, this is handled by a register map.

At this present state, the remote debugger is only developed for a uni-processor system. In a more mature debugging infrastructure, this should be extended to deal with multiprocessors and the problems that this introduces. One problem to deal with, is that debug registers are setup for each processor individually. This means that all processor must have synchronized debug registers to avoid unpredictable behavior. Further, when a processor encounters a breakpoint, all processors should be stopped to avoid that execution continues. If execution continues on the other processors, they may alter data structure and variables and make it difficult to rely on the examined data. To provide this, Advanced Configuration and Power Interface (ACPI) must be implemented, to enumerate the processors available [Hew, 2009]. Synchronization methods to ensure mutual exclusion to shared data structures in the
infrastructure should also be implemented to provide a safe environment for multicore debugging.

RDebug relies highly on GDBParser and is constructed with the GDB protocol basis. If the split between RDebug and GDBParser should have any benefit, RDebug should have used a generic parser interface. This interface could allow different parsers to be used for remote debugging.

User space debugging

The user space debugging interface only provides a thin layer between a debugger and a debugged process’ debug registers. This is a very minimalistic approach but as FenixOS process/thread framework is still very immature, this is sufficient at this point in time. To create a more convincing test scenario, a prototype keyboard driver could have been implemented. This could create a test session, where a debugger could single step another thread by pushing a specific key.

At this point in time, no additional security mechanisms has been implemented. When the process/thread framework gets extended, such security mechanisms should be implemented. In traditional UNIX systems, this is done through the parent/child process relation or by voluntarily wait for debuggers to attach. Other mechanisms like digital signature or cryptographic systems could be used, to ensure that only a specific process may debug a given process.

A side effect of the architecture chosen is the overhead of switching context between every breakpoint or single step, when this is implemented. This is the traditional way of doing this, but new research project tries to limit these context switches. Handlers may be insert directly into the kernel, is one example. This avoids context switches to the debugger process every time a debugged process encounters a breakpoint or single steps [Lin, 2007].

Postmortem analysis

The postmortem facilities provided is a set of panic functions, which in case of a system crash reports necessary information regarding the state to the screen. A in-kernel debugger like KDB could be started when a system crash occurred. This would give a more flexible method of investigating the cause of the fault but would also require interaction. This interaction may change the state of the kernel and leave examination untrustworthy.

Testing strategy

The testing strategy used, has been done by functional testing. This shows that a solution works, but does not show that it will always work. Testing in general
can only show the presence of faults, but more thorough testing strategies exist which may be automated. Testing operating systems can be a difficult task, as some tests may or should crash the system. If this is disregarded, unit testing could be implemented. This testing strategy tests isolated parts of software individually and can be used to ensure that the complete product satisfies its specification. It also helps tracing potential defects, as tests should target a small area of the entire code [Andy Hunt, 2004]. Unit testing makes it possible to use the test driven software development paradigm. In this paradigm, tests are written before implementation of actual code is done. The initial test should of course fail, however as the developer progresses the test should pass. This introduces a very short development cycle and gives means to produce high quality code.
The domain of software defects, infections and faults have been covered to motivate the need for debugging facilities. The complexities of debugging operating systems have been covered and the need for an debugging infrastructure for FenixOS motivated. This thesis has investigated the options for remote debugging of kernel code, hardware support for debugging, user space debugging interface and postmortem analysis. The selected options has been motivated and these choices makes basis for the design presented. This design has been implemented in a prototype implementation which enables remote debugging with GDB, debugging facilities to control user space processes and methods for kernel panic.

The overall reliability and quality of the solution implemented is acceptable and provides a new set of functionalities for kernel debugging FenixOS. This should help developers to decrease the amount of time spent on tracing defects and make it more pleasant job to work with kernel code.
Bibliography


