Norwegian University of
Science and Technology

Master Thesis

Spring 2019

# Strict Memory Protection for Microcontrollers

**Erlend Sveen**

Supervisor: Jingyue Li
Co-supervisor: Magnus Själander

Sunday 17th February, 2019

# Abstract

Modern desktop systems protect processes from each other with memory protection. Microcontroller systems, which lack support for memory virtualization, typically only uses memory protection for areas that the programmer deems necessary and does not separate processes completely. As a result the application still appears monolithic and only a handful of errors may be detected.

This thesis presents a set of solutions for complete separation of processes, unleashing the full potential of the memory protection unit embedded in modern ARM-based microcontrollers. The operating system loads multiple programs from disk into independently protected portions of memory. These programs may then be started, stopped, modified, crashed etc. without affecting other running programs. When loading programs, a new allocation algorithm is used that automatically aligns the memories for use with the protection hardware. A pager is written to satisfy additional run-time demands of applications, and communication primitives for inter-process communication is made available.

Since every running process is unable to get access to other running processes, not only reliability but also security is improved. An application may be split so that unsafe or error-prone code is separated from mission-critical code, allowing it to be independently restarted when an error occurs. With executable and writeable memory access rights being mutually exclusive, code injection is made harder to perform.

The solution is all transparent to the programmer. All that is required is to split an application into sub-programs that operates largely independently. An added benefit of this modularity is that code may also be developed and tested as independent programs. The standard POSIX API is used as the programming interface, allowing programmers to use existing knowledge and code when designing applications. For programs that do not depend on specific hardware, development may be done and tested on a regular desktop system entirely before running it on a microcontroller.

# Acknowledgements

Thanks to supervisors Jingyue Li and Magnus Själander for helping with all the formalities surrounding the thesis, and giving valuable feedback on the thesis work. The work presented in this thesis would not have been possible without supervision.

Also thanks to the students at the Orbit satellite program, which have helped by providing an environment for the thesis in the first place. The satellite program was in need of someone to work on the software, which resulted in both the pre-thesis project and the thesis itself. Students working on the satellite software has provided valuable feedback when using the operating system, helping to improve it.

# Contents


**List of Figures** viii

**List of Tables** ix

**1. Introduction** 1
- 1.1. Research Drivers . . . 1
- 1.2. Solutions . . . 2
- 1.3. Testing and Evaluation . . . 3
- 1.4. Summary . . . 3
- 1.5. Thesis Structure . . . 3

**2. Background** 5
- 2.1. Memory Protection in Desktop Processors . . . 5
  - 2.1.1. Overview . . . 5
  - 2.1.2. History . . . 6
  - 2.1.3. Memory Architecture As Seen By Software . . . 7
  - 2.1.4. Challenges . . . 8
- 2.2. Memory Protection in Microcontrollers . . . 10
  - 2.2.1. History . . . 10
  - 2.2.2. Microcontroller Memory Protection . . . 10
  - 2.2.3. Table Size and Table Entry Restrictions . . . 11
  - 2.2.4. Allocation Example . . . 11

**3. Related Work** 13
- 3.1. State of the Art . . . 13
- 3.2. Shared vs Non-Shared Memory Architecture . . . 15
  - 3.2.1. Microcontroller Shared Memory Problem Example . . . 17
  - 3.2.2. MPU Setup . . . 17
  - 3.2.3. Shortcomings . . . 18
  - 3.2.4. Privilege Escalation . . . 18
  - 3.2.5. Summary . . . 19
- 3.3. Research Motivation . . . 20
  - 3.3.1. Previos Project Work . . . 21

**4. Project Scope** 23
- 4.1. Problem Summary . . . 23
  - 4.1.1. Memory Protection Schemes . . . 23
  - 4.1.2. Key Difficulties . . . 23

4.1.3. Benefits . . . 24
4.1.4. Hardware Popularity and Generalization . . . 24
4.2. Research Questions . . . 25
4.2.1. Question 1: How may memory contents be protected during various failure modes? . . . 25
4.2.2. Question 2: How may the improvements be made without harming performance? . . . 25
4.2.3. Summary . . . 26
4.3. Solution Roadmap . . . 26
4.4. Architecture Overview . . . 27

**5. Memory Allocation** **29**
5.1. Allocation Rules . . . 29
5.1.1. Allocation Rule Summary . . . 30
5.2. Metadata Structures and Multibanking . . . 31
5.3. Allocation Algorithm . . . 33
5.3.1. Algorithm Without MPU Support . . . 33
5.3.2. Algorithm With MPU Support . . . 36

**6. Program Loading** **41**
6.1. Program Structure . . . 41
6.2. The ELF file format . . . 42
6.3. Relocation . . . 44
6.4. Load API Selection . . . 45
6.5. Load Procedure . . . 46
6.6. Context Switching . . . 48

**7. Adding Paging** **50**
7.1. Paging without Virtualization . . . 50
7.2. Adding Paging . . . 50
7.2.1. Architectural Considerations . . . 51
7.2.2. Control Structures . . . 52
7.2.3. System Calls . . . 53
7.2.4. Page Fault Handler . . . 57
7.2.5. Table Walking Algorithm . . . 58
7.2.6. Replacement Policy . . . 60
7.2.7. Miscellaneous Page Table Functions . . . 60

**8. Testing and Verification** **62**
8.1. Memory Allocation . . . 62
8.1.1. Performance Considerations . . . 62
8.1.2. Functionality Testing . . . 63
8.1.3. Test Subjects . . . 64
8.1.4. Methodology . . . 64

8.1.5. Expected Results . . . . . . . . . . 65
8.1.6. Test Execution . . . . . . . . . . 66
8.1.7. Discussion . . . . . . . . . . 66
8.1.8. Conclusion . . . . . . . . . . 68
8.2. Program Loading . . . . . . . . . . 68
8.2.1. Performance Considerations . . . . . . . . . . 68
8.2.2. Test Subjects . . . . . . . . . . 69
8.2.3. Methodology . . . . . . . . . . 69
8.2.4. Expected Results . . . . . . . . . . 71
8.2.5. Test Execution . . . . . . . . . . 71
8.2.6. Discussion . . . . . . . . . . 72
8.2.7. Conclusion . . . . . . . . . . 72
8.3. Paging and Memory Protection . . . . . . . . . . 73
8.3.1. Performance Considerations . . . . . . . . . . 73
8.3.2. Test Subjects . . . . . . . . . . 74
8.3.3. Methodology . . . . . . . . . . 74
8.3.4. Expected Results and Test Execution . . . . . . . . . . 76
8.3.5. Discussion . . . . . . . . . . 78
8.3.6. Conclusion . . . . . . . . . . 79
8.4. Small System Use Case Example . . . . . . . . . . 79
8.4.1. Methodology . . . . . . . . . . 79
8.4.2. Test Execution . . . . . . . . . . 80
8.4.3. Results . . . . . . . . . . 81
8.4.4. Discussion and Conclusion . . . . . . . . . . 84
8.5. Memory Copy Speed Test . . . . . . . . . . 85
8.5.1. Methodology . . . . . . . . . . 85
8.5.2. Expected Results . . . . . . . . . . 85
8.5.3. Test Execution . . . . . . . . . . 86
8.5.4. Discussion and Conclusion . . . . . . . . . . 86
8.6. Added Context Switching Overhead . . . . . . . . . . 86
8.7. System Call Interface Overhead . . . . . . . . . . 87
8.7.1. System Calls . . . . . . . . . . 87
8.7.2. Benefits and Limitations . . . . . . . . . . 88
8.7.3. Performance Study . . . . . . . . . . 88
8.7.4. Summary . . . . . . . . . . 92

**9. Discussion** **94**
9.1. Solution Discussion . . . . . . . . . . 94
9.1.1. Memory Allocation . . . . . . . . . . 94
9.1.2. Program Loading . . . . . . . . . . 94
9.1.3. Paging Mechanism . . . . . . . . . . 95
9.2. Research Question Discussion . . . . . . . . . . 95
9.2.1. Research Question 1 . . . . . . . . . . 95
9.2.2. Research Question 2 . . . . . . . . . . 96

**10. Conclusion** **98**

**11. Future Work** **99**
11.1. Memory Allocation Improvements . . . . . 99
11.2. Paging Improvements . . . . . 99
11.3. Memory Sharing Schemes . . . . . 99
11.4. Improved Standards Compliance . . . . . 100
11.5. Hardware Error Handling . . . . . 100
11.6. Syscall Data Access Checking . . . . . 100
11.7. Moving Data Back Into Flash . . . . . 100
11.8. Test Procedures . . . . . 101
11.9. Build Automation . . . . . 101
11.10. Power Management . . . . . 101
11.11. Kernel Security . . . . . 101
11.12. General Operating System Development . . . . . 101

**References** **102**

**Appendices** **104**

**A. Supplied Archive** **105**
A.1. Supplied Archive Structure . . . . . 105
A.2. Code Licensing . . . . . 106

**B. Code Listings** **107**
B.1. Memory Allocation Test Code . . . . . 107
B.2. Shell Function For Program Execution . . . . . 108
B.3. Crash Test Code . . . . . 109
B.4. Example Use Case Test Code . . . . . 113
B.5. `kernel/kernel/src/mem/mem.c` . . . . . 120
B.6. Cortex-R Context Switch . . . . . 129

# List of Figures

2.1. Overview of the address translation process . . . . . . . . . . . . . . . . . . 6
2.2. Overview of the address translation in a multi-process system . . . . . . . 8
2.3. Memory allocation regions available . . . . . . . . . . . . . . . . . . . . . . 12
2.4. Memory allocation sub-regions available . . . . . . . . . . . . . . . . . . . . 12

3.1. Desktop vs. microcontroller memory systems . . . . . . . . . . . . . . . . . 16

4.1. Overview of the system architecture . . . . . . . . . . . . . . . . . . . . . . 27

5.1. Fragmented memory due to allocation metadata . . . . . . . . . . . . . . . 30
5.2. Memory fragmentation avoided by moving metadata . . . . . . . . . . . . . 30
5.3. Example of the memory list layout in memory . . . . . . . . . . . . . . . . . 32
5.4. Simplified Region Base Address Register . . . . . . . . . . . . . . . . . . . . 36
5.5. Simplified Region Attribute and Size Register . . . . . . . . . . . . . . . . . 36

7.1. Linux Page Table Layout[1] . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

8.1. Memory Allocation Test Progress . . . . . . . . . . . . . . . . . . . . . . . . 67
8.2. Testing the process loading on the desktop . . . . . . . . . . . . . . . . . . . 71
8.3. Testing the process loading on the microcontroller . . . . . . . . . . . . . . 72
8.4. Example Application Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . 80

# List of Tables

3.1. Comparison of advantages in shared and non-shared memory systems . . 15
3.2. Allocated memory protection regions . . . . . . . . . . . . . . . . . . . . . . 18

# List of Algorithms

1. Top-level algorithm outline: MemAlloc . . . . . . . . . . . . . . . . . . . . . 34
2. Entry allocator: MemAllocEntry . . . . . . . . . . . . . . . . . . . . . . . . 35
3. Memory deallocator: MemFree . . . . . . . . . . . . . . . . . . . . . . . . . 35
4. Top-level algorithm outline: MemAllocMPU . . . . . . . . . . . . . . . . . . . 40

5. Simplified process loading procedure . . . . . . . . . . . . . . . . . . . . . . 47

6. Memory management: SvcMemmanHandleMmap . . . . . . . . . . . . . . . 55
7. Memory management: SvcMemmanHandleMunmap . . . . . . . . . . . . . . 57
8. Memory management: PagerWalkHandle . . . . . . . . . . . . . . . . . . . . 60

# List of Listings


1. The dangerous `vTaskDelay` function . . . . . . . . . . . . . . . . . . . . . . 19
2. Privilege escalation function . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3. The memory bank structure . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4. The structure used for MPU allocations . . . . . . . . . . . . . . . . . . . . 37
5. The structures of the ELF file format . . . . . . . . . . . . . . . . . . . . . 43
6. Example structure of a program . . . . . . . . . . . . . . . . . . . . . . . . . 44
7. The `posix_spawn` function prototype . . . . . . . . . . . . . . . . . . . . . 46
8. The beginning of the `ThreadControl` structure . . . . . . . . . . . . . . . . 49
9. Restoring the MPU registers in assembly . . . . . . . . . . . . . . . . . . . . 49
10. The page table structures . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
11. The memory mapping functions [2, 3] . . . . . . . . . . . . . . . . . . . . . . 53
12. The PagerAddEntry and PagerCalculateMaxAddress functions . . . . . . 56
13. The `DataAbort` function . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
14. The `PagerWalk` function . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
15. The C copy function . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
16. C Library implementation of the `write` function . . . . . . . . . . . . . . 89
17. Assembly handler for the `svc` instruction . . . . . . . . . . . . . . . . . . 89
18. Kernel mode write call handler . . . . . . . . . . . . . . . . . . . . . . . . . 90
19. Virtual file system write call handler . . . . . . . . . . . . . . . . . . . . . 91
20. Device file system write call handler . . . . . . . . . . . . . . . . . . . . . 91
21. Write function of `/dev/null` . . . . . . . . . . . . . . . . . . . . . . . . . 92

# Glossary

**ABI** Application Binary Interface. 27

**API** Application Programmer Interface. 19, 26, 27, 45, 46, 53, 70, 94–96, 98–100

**CCM** Core-Coupled Memory. 31

**CIA** Confidentiality, Integrity, Availability. 14

**CPU** Central Processing Unit. 36, 42, 43

**DFAR** Data Fault Address Register. 77

**DFSR** Data Fault Status Register. 59

**DMA** Direct Memory Access. 99

**ECC** Error Correcting Code. 22

**ELF** Executable and Linkable Format. 42

**GCC** GNU Compiler Collection. 19

**GNU** GNU is Not Unix. 42

**IO** Input-Output. 69, 75

**IoT** Internet of Things. 24

**ISR** Interrupt Service Routine. 19

**MMU** Memory Management Unit. 11, 46

**MPU** Memory Protection Unit. 2, 10, 11, 13, 14, 16–18, 20, 21, 23, 24, 26, 28, 29, 33, 35–38, 48–54, 57, 59, 63–68, 73, 75, 78, 86, 87, 97

**OS** Operating System. 27

**PGD** Page Global Directory. 51

**PID** Process ID. 46, 48, 52, 54, 58, 61, 74

**PMD** Page Middle Directory. 51

**POSIX** Portable Operating System Interface. 2, 13, 45, 46, 53, 69, 72, 94, 98

**PTE** Page Table Entries. 51

**RAM** Random Access Memory. 18, 31, 75

**RASR** Region Attribute and Size Register. 36, 38, 53, 55, 64, 67

**RBAR** Region Base Address Register. 36, 38, 49, 67

**SRD** Sub-Region Disable bits. 36

**SVC** Service Handler. 27

**TLB** Translation Lookaside Buffer. 5

# 1. Introduction

In modern desktop and high-end embedded systems, application address spaces are separated from each other using memory protection hardware. The protection mechanisms are managed by the operating system, and applications may explicitly agree to share memory when desired. Hardware registers are only accessible to drivers that are given access by the Operating System. The hardware implements memory virtualization that enables each process to see the entire processor address space as its own, preventing them from accessing other applications memories. Paging is used to split the memory into smaller blocks called pages, which are assigned by the Operating System. Combined with virtualization, pages may be allocated, freed, moved, and swapped to slower storage as desired.

Microcontrollers are devices that embed the processor, memories and peripherals in a complete and small package. They power a large range of applications from alarm clocks to automotive entertainment systems and more. Due to their history in simple applications, memory management has been simple and memory protection unnecessary. As technology has evolved, however, both the microcontrollers and the software they run has grown. Modern microcontrollers typically embed a Memory Protection Unit, that enables protection for a few selected memory areas. Due to hardware complexity and cost, virtualization is typically not supported, and memory management is tedious. As a consequence protection is only used to protect programmer selected memory areas: good enough to detect serious faults, but not good enough to guarantee that a system will remain stable after a fault. Data loss is still likely, and due to the effort required to leverage the hardware, the protection features often remain unused.

## 1.1. Research Drivers

There are two main questions driving the research presented in this thesis. The first is about memory protection itself: how may the contents of a memory be protected during different failure modes? A memory may not just be different physical memories, but also logical blocks of memory within the same physical devices. Computer systems experience many different failures, with causes ranging from simple programmer errors to intentional security breaches or even hardware failures. The second research question relates to performance. How may improvements be made without harming performance? While performance impact may be measured, it is up to the application programmer to decide the acceptable limits of performance loss.

In the world of embedded operating system software, an overwhelmingly large majority of the systems that are certified to any notable safety standards are also proprietary and much too expensive for most applications. During the research phase of this thesis, no

open source operating system that implements a restrictive memory protection scheme while also fitting inside the internal memories of the typical microcontroller were found, at least not in marketing material. Strict protection features appear to be a feature reserved for larger systems.

It is assumed that strict memory protection for Memory Protection Unit (MPU)-enabled microcontrollers with less than a megabyte of memory is a new addition especially in the world of open-source software. This thesis continues with an implementation of a strict memory scheme using techniques borrowed from the desktop world, adapted for use in small systems where memory constraints, time to market, and predictability are important.

The work presented is mainly built upon techniques found in major open-source software such as the Linux kernel and open-source standards such as Portable Operating System Interface (POSIX) [4]. Using standards for application interfaces enables programmers to use existing knowledge, allowing for much quicker adaptation of the solution. It is also the intention for the solution to remain open-source so that it will be available to anyone that desires to use it.

## 1.2. Solutions

This thesis presents a solution that enables programs running on a microcontroller to be completely separated without added developer effort. In chapter 5, a memory allocator and deallocator is invented that manages memory at run-time while respecting the challenging limitations of the hardware. This allocator is the fundamental building block that enables the hardware protection features to be fully utilized at all times. Developers do not have to concern themselves with address calculations, alignment and register values any longer since the allocator handles everything seamlessly. This allocator provides the foundation for the other subsystems that build upon it, and makes it possible to answer the first research question.

A program loader is written and then presented in chapter 6 which is able to load multiple programs into memory and starts their simultaneous execution, using the allocator to ensure that they are separated. This ensures that the allocator and memory protection features are used at all times. By doing this at run-time, programs may be started, stopped, changed etc. at any time without interfering with other programs. Most importantly it enables a program to crash without compromising the system and other applications integrity. A monitoring program may then decide to act on the program crash and handle the error, for instance by restarting the problematic program and alerting the user. The loader follows the POSIX standard where possible providing a known programming interface for both starting programs and managing them.

At last a paging mechanism is invented which enables running software to allocate additional protected memory at run-time. The solution pushes the protection hardware to its limit by using free protection registers to implement a manually managed paging scheme. Since the pager pushes beyond what most microcontroller hardware is intended for, a very simple paging scheme is presented, sacrificing performance for simplicity.

Using the pager is made completely optional ensuring that no performance is lost when it is not used.

## 1.3. Testing and Evaluation

Once all the sub-systems are described, testing and evaluation commences. The goals of the functional and performance testing is to assess how well the implementation satisfies the research questions, and how improvements may be made with continued work.

Performance of the various systems are evaluated with mathematical analysis, functional testing and discussion of the selected solution(s). The allocation subsystem is evaluated by looking at its time complexity properties, which is how it scales with the number of allocations. The design of the loader is evaluated by looking at its time characteristics. At last, some basic performance tests are made to see how the system fulfills its design goals and how it performs in practice.

## 1.4. Summary

Evaluation of the solutions show that the system is indeed able to accomplish strict memory protection with a limited performance impact. Software error may occur in one program without affecting unrelated programs and the system remains stable. Performance depends on the characteristics of the application: those that spend most of the time computing local data will experience very little performance loss, while those that make many system calls will experience more. This performance loss is inherent to the system call interface used as the bridge between individual programs. Programs making frequent system calls are tested to experience a 21 % to 28 % performance penalty depending on use of the pager.

The end solution is suitable for applications with needs that fall between high end microcontrollers and low end embedded Linux systems: where the basic features of a Linux system is often desired, but the hardware cost is not justifiable. By leveraging the solutions presented in this thesis high end microcontrollers will be able to serve these applications in a convenient and robust manner.

## 1.5. Thesis Structure

The thesis is structured into three main portions: getting a good understanding of the background material, the solutions themselves, and at last testing and evaluation of the presented solutions.

The first chapter is intended to bring a fundamental understanding of memory protection in current, competing and historical systems. This is contained in chapter 2, the background chapter.

Next, the related work chapter documents current software systems and their limitations. Since most of the systems that are likely to implement solutions comparable to the ones presented in this thesis are closed systems, and very expensive, making the scope

of this background research somewhat limited. It is in any case assumed that an open source implementation is a great contribution to the world of microcontroller software, and so the work presented in this thesis is also open.

Chapter 4 formally describes the questions driving the research in the thesis. These are then referenced in the testing chapter in order to check how well they have been satisfied.

The second portion of this thesis are the chapters describing the solutions: Memory allocation in chapter 5, program loading in chapter 6 and at last paging in chapter 7. These all build upon each other creating an easy to use system that enforces strict memory protection between processes.

The last main portion of the thesis consists of testing, verification, evaluation and discussion of the solutions. Testing is mainly done to verify that the systems work according to intention and to check against the driving research questions. The tests also function as a showcase of what the system is able to accomplish in practical usage. At last a general discussion is made, potential improvements are suggested and a conclusion is made.

# 2. Background

In this section, we introduce the solutions employed in current systems. The various memory schemes used in desktop, mobile and microcontroller systems are briefly discussed. The challenges related to memory protection in microcontrollers are then discussed.

## 2.1. Memory Protection in Desktop Processors

In desktop computer systems, multiple programs are run at the same time sharing the processing units and available memories. Separating the memory accessible to these programs improves the stability, reliability and security of the system. This section provides a brief overview of how this is accomplished and the related challenges.

### 2.1.1. Overview

In modern desktop processors, process separation is achieved through memory virtualization and paging [5]. These two technologies are often used together but must not be confused: paging is a memory allocation scheme where memory is organized into blocks called pages, while virtualization is the translation of memory addresses.

When the modern processor needs to access memory the address is checked against a page table, which is cached by the processor in a Translation Lookaside Buffer (TLB). This table contains translation entries for translating virtual addresses that the processor uses to real memory addresses where the data is. With this technique each process sees the entire processor address space as its own, free of conflict, since the table is also swapped along with the process. Sharing memory can be achieved through setting an entry in multiple programs to the same physical address. Page sizes on modern x86 systems are usually a mix of 4KiB, 1MiB and even 1GiB.

Each entry in the table is marked as residing in real memory or not. If the running program accesses memory that is not located in real memory, the access triggers an exception that invokes the page supervisor. The page supervisor is part of the operating system and is responsible for fetching the data into memory from an indirectly accessible medium such as a hard drive, changing the table entry and resuming the program execution. With this technique it is possible to “simulate” more memory than what is physically installed. A similar exception occurs when the processor does not have the correct entry cached in the TLB. In those cases the supervisor will have to fetch the entry for it.

If the memory access was invalid, the operating system will not find the correct page neither on disk nor in main memory. An exception is then raised so that the offending

program may handle the error gracefully or terminate.

In addition to virtualization the page table entries also contain the access rights of the page. Typical access rights are execute, read and write flags. Violating these rights will also trigger an exception similar to accessing invalid pages, allowing the fault to be handled.

A simplified schematic of how the translation is performed is illustrated in figure 2.1. In this case the user program loads the address `0001234h` into register `R3` and loads the data word located at that address into register `R2`. Since the memory is virtualized, the address specified is not used directly. Instead, the address is divided into two parts: the page index and the page offset. The number of bits used in each directly correlates to the page size used which in this case is 4KiB. This gives the 12 bits for the offset and 20 bits for the page index which is used as input into the table. The physical page index is then taken from the table and used to assemble the final physical address with the original offset. Simultaneously, the access flags in the table is checked for any violations. In this example only the valid flag is shown.

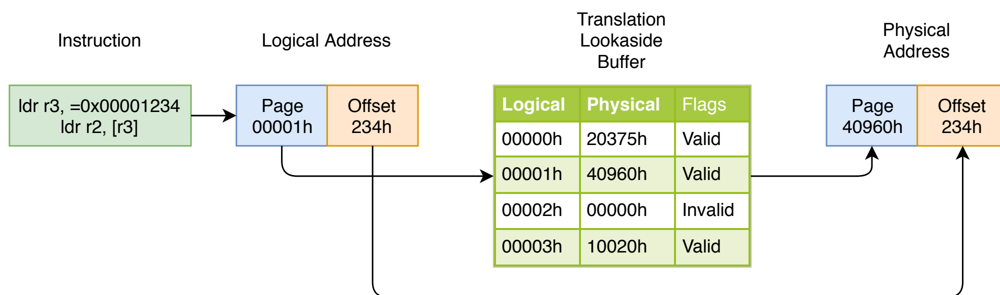


Figure 2.1.: Overview of the address translation process

### 2.1.2. History

Memory virtualization goes back to the mainframe era of computing, when the main memory was composed of very expensive core memory. The size of these memories was in the thousands of words size range. Memory virtualization was then introduced as a means of simulating more memory through disk swapping, allowing machines configured with less memory to run more demanding programs. The NORD-1 made by Norsk Data was the first minicomputer to offer memory virtualization in the early 70s [6].

For the x86 processor architecture that is used in personal computers today, virtualized memory was first introduced with the Intel 80286 processor. The design used memory segmentation where each memory address was composed of a segment index and an offset into that segment. Multiple segments were used in the processor for program text, data, stack and 'extra' segment.

The Intel 80386 introduced paging in addition to segmentation. This added an additional level of address translation from the segmentation addresses to the real addresses providing virtualized memory like it is known today. With the 64-bit version of the

x86 architecture, the segmentation is removed, providing a flat memory space with only paging and address translation.

### 2.1.3. Memory Architecture As Seen By Software

With virtualized memory, each process running on the system has its own address space. It can not see any data belonging to other processes. In order to communicate with the outside world the process uses system calls, a kind of software interrupt, to make the operating system perform simple tasks on behalf of the process. Examples of such tasks are reading key presses from the keyboard, writing a file or sending a packet on the network.

The operating system on the other hand has components that always operates in a privileged state. These software components are allowed to access all memory, which is necessary in order to perform system calls on behalf of processes among other tasks. The core component of the operating system which performs these tasks is often called the Kernel. The Kernel is responsible for starting the system and provide all the basic communication and cooperation primitives to build a functioning system. Examples are processor time scheduling, memory management, device drivers and file systems. Kernels are carefully written so that a process with malicious intent can not gain access to objects it should not have access to.

In some cases it is beneficial for programs to share memory directly, allowing communication to take place at the highest performance. The processes involved will then set up a shared piece of memory with the help of the Kernel, and the Kernel will then set entries in the page tables of both programs to the same piece of physical memory.

This kind of memory sharing is not always explicit. Since many programs often share the same code libraries, such as the standard C library and various graphics libraries, code segments may be shared between processes. In order to prevent one program from affecting another this code is then set to read-only execute permissions. The Kernel may also employ a copy-on-write scheme where an actual copy of the data is only made when the process with the intended copy does its first write. This approach helps making some operations significantly faster, such as ‘fork’.

Desktop memory architecture is, therefore, non-shared by default, and sharing is then applied when desired. This may also be referenced to as a mixed memory architecture.

Figure 2.2 illustrates how the address translation takes place in a multi-process system. Here, two identical programs are run in parallel, each thinking it has the processor as its own using the same addresses. In reality the page table has different entries for the two program instances pointing the memory accesses to different parts of memory, allowing these initially conflicting programs to coexist.

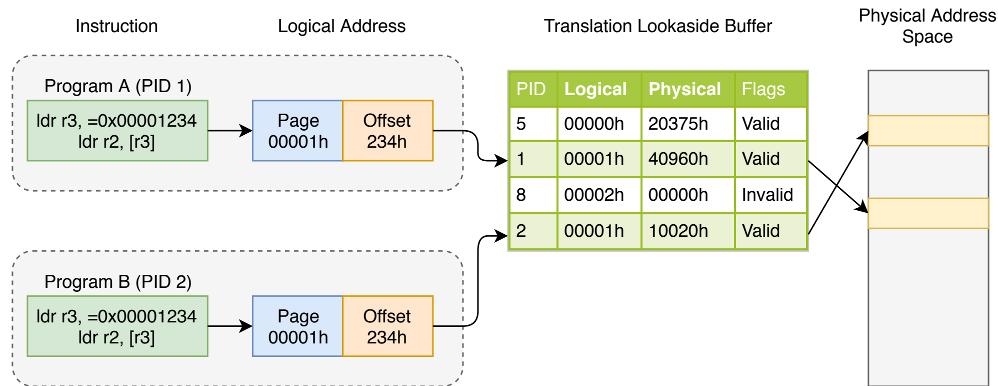


Figure 2.2.: Overview of the address translation in a multi-process system

### 2.1.4. Challenges

In this section the quirks and challenges associated with paged memory are briefly discussed.

#### Memory Fragmentation

Memory fragmentation is a problem that may appear in systems where memory is frequently allocated and freed in various sizes, causing small fragments of free memory to reside between blocks that are in use. If these fragments can not be collected into continuous blocks when needed, new allocations may fail despite having enough memory since there is no continuous block large enough to satisfy the request.

The process of collecting these fragments is called *defragmentation*. In systems without virtualized memory, free or used blocks cannot be moved without direct cooperation with the programs that uses them. In systems *with* virtualized memory, physical memory may be defragmented by copying data to a new location and changing the translation in the page table. In the virtual address space holes may be present without any actual memory usage by simply not having entries in the page table for it. This memory is often referenced as reserved or committed memory by software tools.

Virtualization and paging provide a great deal of flexibility when dealing with fragmentation issues. For systems that support multiple page sizes, smaller pages may even be combined into larger ones for smaller page tables and more efficient cache usage.

#### Not Enough Memory

As previously discussed, having a small main memory may be compensated for by swapping pages to slower but larger storage. Swapping enables computers with too little memory to complete its task at the cost of performance. Disk swapping requires a form of virtualized memory with paging allowing the swapping to appear transparently to the affected program.

Without virtualization, the capacity of the computer's main memory will impose a hard limit on the software memory usage. Since memory fragmentation is likely to occur without virtualization, out-of-memory errors may even occur before the memory resources are exhausted.

### Non-Deterministic Performance

With paged virtualized memory the translation vectors has to be stored somewhere. Since the page tables are often large, the processor itself only contains a small cache of the most recently used entries. This cache is called the Translation Lookaside Buffer (TLB). When a program accesses some memory that is not in the cache the operating system is invoked so that it may fill the table with the correct entries. Fetching these entries takes time. If the requested memory was in real memory but not in the cache, the penalty may only be in hundreds of cycles. If the memory was swapped to disk however the penalty may be in millions of cycles, as hard drives have to position the read and write mechanism physically on the platters before reading data.

This introduces some randomness into the fine timings of the programs as the times when these events occur are hard to predict. In larger systems where bulk computing performance is what matters and some timing jitter is accepted, other benefits outweigh that of deterministic performance.

Some systems, such as the Xilinx Zynq UltraScale+ [7], combine processors with and without virtualized memory to gain the benefits of both. Timing sensitive software is then run on the processor without virtual memory while general higher-throughput computing is done with processors that offer virtual memory.

### Page and Disk Thrashing

Thrashing is a phenomenon that happens when a page is evicted from either the page cache or main memory only to be brought back in again shortly after. The underlying cause is if the cache or memory is way too small causing most of the processing time to be spent on solving page faults.

Such problems are usually solved by upgrading the machine with more memory or a larger page cache. In some cases the running program may be altered so that its memory accesses are more suitable, however this is often not possible or too difficult. Thrashing often indicates that a machine is running programs with requirements beyond its capabilities.

## 2.2. Memory Protection in Microcontrollers

In this section, the memory protection hardware found in microcontrollers is discussed briefly. This provides a basis for comparison against the desktop methods and an understanding of its limitations.

### 2.2.1. History

The first microprocessors developed in the early 70s required multiple external components to function, such as memories and input-output devices. The increased cost made the microprocessor unsuitable for use in many simple appliances.

The microcontroller integrates a simple processor and all the required components in a single chip. A typical configuration in an early chip would be an 8-bit processor, a few kilobytes of read-only program memory and a handful of bytes for the working memory. This small configuration is sufficient for simple applications such as alarm clocks, microwave ovens and computer keyboards. The 8-bit microcontrollers still hold a significant market share, as they are the simplest to produce for use in simple appliances.

Other uses, such as audio/video processing and advanced control systems in cars and industry equipment has provided a market for more advanced microcontrollers as well. In 2011, the market share of 16-bit microcontrollers grew past the 8-bit market share for the first time [8]. Increased demand for Internet connectivity in many applications as well as intelligent car systems has also given rise to 32-bit microcontrollers. These 32-bit parts share many architectural aspects with their desktop counterparts, with the main differences now being computer power and memory capacity.

The early microcontroller applications did not require any memory protection or special safety hardware due to the low software and application complexity. With the increased use in automobile, industrial and internet-enabled applications, attention has shifted towards security-, reliability- and safety-features. As desktop and embedded processors converge, microcontroller processors has been enhanced with basic memory protection features.

### 2.2.2. Microcontroller Memory Protection

Memory protection in Cortex-M [9] and Cortex-R [10] microcontroller processors are implemented in a special MPU. The MPU monitors all memory accesses done by the CPU and checks that they are legal by looking in a table initialized by the programmer. Each table entry contains the address range and access rights of a memory “region”. The memory address to be accessed is used as the table key, and the access type is compared to the table entry. The access type specifies if the access is part of an instruction fetch or a regular data load/store, and the privilege level of the access. If the table entry is in conflict with the access type, an instruction abort exception is triggered in case of an instruction fetch, or a data abort exception is triggered in case of data accesses.

### 2.2.3. Table Size and Table Entry Restrictions

The table inside the MPU has a fixed size which depends on the options selected by the silicon manufacturer. Typical configurations are either 8, 12 or 16 table entries. The application programmer may choose to use less than the available number of entries, however if more is desired, a software mechanism that swaps regions on an abort exception has to be implemented.

Furthermore, each table entry has limits on the size and alignment of a memory region. The MPU hardware used in Cortex-M/R processor requires that the size of the region is in a power of two and that the base address of the region is aligned to the same power of two. Each region has eight sub-regions that is one-eight the size of the full region, each with a sub-region disable bit, allowing some flexibility during region allocation.

The most significant difference between the MPU and the desktop Memory Management Unit (MMU) counterpart is the lack of virtualized memory. That is, there is no translation of addresses allowing each process to see its own memory address space. They all have to coexist within the same address space, and must not be dependent on being located at a specific address.

As discussed in section 2.1.4 virtualized memory may be used to simulate more memory via disk swapping. In embedded systems without virtualized or paged memory, it is not possible to pretend to have more memory than one already has without extreme performance penalties. When the software reaches the point where the main memory is full or too fragmented no further requests can be fulfilled. The software will then have to manage its memory more carefully or terminate.

It is also not possible to defragment memory after allocation, or to have smaller fragments appear as a continuous page by using multiple translation entries as discussed in section 2.1.4. The effect is that software may run out of memory despite enough being technically available, since there is no single portion large enough to fulfill the request.

As discussed in section 2.1.4, virtualized memory is likely to add randomness to the fine timings of a program. Embedded software that has to respond and act on interrupts within a small set of machine cycles can not accept such non-deterministic performance, which is one of the reasons why virtualized or paged memory is not used for such systems.

### 2.2.4. Allocation Example

In this section, an example is provided for an allocation of 35000 bytes. A memory allocation of 35000 bytes requires a region allocated to the nearest power of two, rounded up. This resolves to $2^{16} = 65,536$ bytes, an over-allocation of $30,536$ bytes or 87%. The eight sub-region disable bits may be used to reduce the amount of wasted space. With a region size of $2^{16}$ bytes a sub-region is $2^{13} = 8192$ bytes. Now using the sub-region size, a total of five continuous sub-regions is required to fulfill the request. The allocation is now $40,960$ bytes, an over-allocation of 17%.

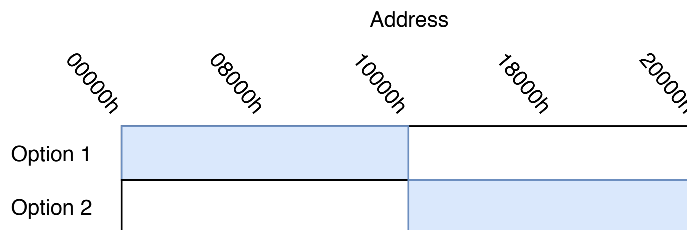


Figure 2.3.: Memory allocation regions available

Figure 2.4.: Memory allocation sub-regions available

Figure 2.3 illustrates the two regions that may be used for a region size of $2^{16}$ bytes and a total memory size of $2^{17}$ bytes. The blue marked area represents a region while the white area is untouched memory. Figure 2.4 illustrates the set of working sub-region configurations that may be used, enhancing the detail of figure 2.3. The blue fill is replaced by eight blocks representing the sub-regions where green is the active sub-regions and red is the disabled sub-regions. The allocation algorithm must then find a free memory block that can be protected using any one of these configurations. In order to prevent unintended access from later allocations, the entire $40,960$ byte block must be reserved despite only needing $35,000$ bytes.

As made clear by figure 2.4 there are some missing combinations, where the green blocks would span the center of the memory box. This would however require either two 64KiB regions with the sub-regions split among them, or one 128KiB region spanning the entire memory. The trade-off in the first case is that two regions are used on a system that may only have eight regions available. In the second case, the sub-region size is now twice as large, causing an even greater over-allocation.

# 3. Related Work

Existing operating systems in the desktop world has made heavy use of memory protection schemes since the Intel 386 processor launched, although adaptation was rather slow. Certain systems such as Linux has been built with these features incorporated from the very beginning [11]. Mainframe systems have used such techniques from even earlier on. Aggressive memory protection in itself is, therefore, nothing new. This provides a huge amount of material that may be used to understand the techniques involved and ways to solve the problems of memory management.

Deeply embedded systems is however much different. As detailed later in section 4.1, the hardware necessary for the high-performance solutions often associated with desktop systems is too expensive for deeply embedded systems. The hardware is however able to perform many tasks just like its bigger brothers, but due to the complexity of getting around the hardware limitation, it is rarely used to its fullest.

In the world of embedded operating system software, an overwhelmingly large majority of the systems that are certified to any notable safety standards are also proprietary and much too expensive for most applications. During the research phase of this thesis, no open source operating system that implements a restrictive memory protection scheme while also fitting inside the internal memories of the typical microcontroller were found, at least not in marketing material. Strict protection features appear to be a feature reserved for larger systems. However, systems that takes different approaches to memory protection does exist, such as [12], [13] and [14].

It is assumed that strict memory protection for MPU-enabled microcontrollers with less than a megabyte of memory is a new addition especially in the world of open-source software. This thesis continues with an implementation of a strict memory scheme using techniques borrowed from the desktop world, adapted for use in small systems where memory constraints, time to market, and predictability are important.

The work presented is mainly built upon techniques found in major open-source software such as the Linux kernel and open-source standards such as POSIX [4]. Using standards for application interfaces enables programmers to use existing knowledge, allowing for much quicker adaptation of the solution. It is also the intention for the solution to remain open-source so that it will be available to anyone that desires to use it.

## 3.1. State of the Art

Microcontroller software and tools are still made and used much like in their 8-bit days. By default, a software package is compiled as a monolithic application using trusted software libraries and stored entirely in the internal ROM. The memory protection hardware

is not used in these applications, and in the case of internet connected applications, this may pose a security threat. One example of a product hacked through memory exploits is the hack of the Amazon Dash Button, which allowed code to be injected and executed for a complete memory dump [15]. The hack was done by exploiting a buffer overflow bug that allowed arbitrary fragments of code to be executed from the buffer. By performing a complete memory dump using injected code, an attacker is then able to fully reverse engineer the application and uncover any secrets located in internal memory such as cryptographic keys and any other valuable information.

In applications that require a higher degree of security, reliability or safety, the MPU is used to protect selected sensitive areas in memory. This protects these areas from unintended or unauthorized access and enables any occurrences of such to be handled as any other invalid access and trigger an interrupt. Security (Confidentiality, Integrity, Availability (CIA)) is improved by restricting access by unauthorized code to certain areas. Code that attempts the access may be stopped and the user may be alerted. Reliability is improved as accidental errors are caught and handled. By preventing the fault from escalating into undefined behaviour, important data may be salvaged before a reboot, or the application may even be able to continue normally. Safety is improved in a much similar way by preventing undefined behaviour. In safety-critical applications, unexpected or unknown errors are often handled by entering a special safe mode to prevent any damages.

The end result is a mostly shared memory architecture combined with non-shared memory architecture where desired. This mix is due to the program instructions residing in a single block of memory without strict separation between tasks by default, and then applying access restrictions selectively in desired areas. This use of the MPU is a joint effort between the threading library providing any default protection and the end application programmer setting up additional protection regions as desired. An example is discussed in section 3.2.1. The main advantage of this setup is that errors causing malicious or unintended access to the enabled regions will be caught by the hardware which will trigger an interrupt. By handling this interrupt properly, a graceful restart may be executed or the product may enter a special safe mode.

On the other hand, setting up regions selectively in bare-metal applications requires much more effort by the application programmer for it to have any good effect. In particular, separating stacks and individual instruction memories for different threads pose a big challenge due to the limitations of the MPU as discussed in section 2.2.3. For each group of memory that is to be accessed by the same executing thread with the same privilege, an aligned memory containing the data has to be set up, and the table has to be altered properly when changing threads. If a thread is allowed write access to another threads memory through shared regions, a bug in one may appear as a crash in another or go completely unnoticed. For security applications, even read only access may pose a significant threat. These disadvantages are inherent to shared memory architecture. Developing with explicitly set memory protection involves a trade-off between development cost and benefits.

## 3.2. Shared vs Non-Shared Memory Architecture

In this section, the virtualized memory approach used in desktop processors and the shared memory system typical to microcontrollers is compared directly. The shared memory architecture for the microcontroller used in this comparison is the architecture typically found in FreeRTOS ports, in particular the port for the Texas Instruments TMS570 microcontroller. The virtualized memory approach for the desktop is the type found in Linux and Windows systems.

It is important to note that a shared memory architecture in the microcontroller is typically system wide, and not just between individual programs that explicitly agreed upon the sharing. In the desktop world, each process is given its own non-shared address space using memory virtualization technology, with sharing applied when needed. Note that virtualization is not the same as shared or non-shared memory architecture, but a means to achieve it with relative ease and automation by the operating system.

Table 3.1.: Comparison of advantages in shared and non-shared memory systems

| | Desktop | Microcontroller |
|---|---|---|
| Shared instruction memory between user programs[1] | Yes | No |
| Crash detection rate | Strong | Weak |
| Crash recovery success rate | High | Medium |
| Protection of hardware registers | Yes | Yes |
| TLB/Memory region entries[2] | 64+ | $8 - 16$ |
| Virtualized memory[3] | Yes | No |
| Live memory defragmentation possible | Yes | No |
| Simulation of more memory possible | Yes | No |

[1] Shared memory can be explicitly agreed upon in desktop systems. In some cases, such as with shared software libraries, code memory is shared by the operating system automatically.
[2] Processors store the entries in a cache that is managed by the operating system. Although technically possible in microcontrollers, this is not done.
[3] Enables memory defragmentation and simulating more memory.

In table 3.1, crash detection rate and crash recovery success rate relates to the ability of recovering from bugs causing undefined behaviour. These are typical programmer errors and in the case of memory management a runaway program may end up accessing memory it is not allowed to. Because memory is normally non-shared in the desktop memory architecture, the potential damages are limited. The detection and recovery rate is high. In the typical microcontroller implementation, however, the memory is by default shared. This opens up small windows where a bug may go undetected and cause damage to other programs or tasks.

Protection of hardware registers refers to the ability to protect the hardware accesses itself, which is in direct control of sensors and actuators. In both systems these registers are often the most important asset to protect since modifications will affect the real world directly.

The number of memory region entries is the size of the table physically located in the processor. In desktop systems, this table is a cache for a much bigger table located in main memory. This is not the case for microcontroller systems, usually only a few of the regions are set permanently for the entire run-time of the application and then some are swapped during context switches. Although it is technically possible to use microcontroller hardware as a software-managed cache, this is not done.

Having memory virtualization technology or not is the main architectural difference between desktop systems and microcontroller systems. Although virtualization is not synonymous with shared or non-shared memory architecture, it enables the memory architecture in desktop systems by providing the necessary hardware mechanisms for an efficient implementation. In addition it makes live memory defragmentation and simulation of more memory possible. Despite its usefulness it is not implemented in microcontrollers due to hardware cost and complexity reasons. Most applications does not strictly need it either since they are still of low complexity when compared to most desktop software.

Many embedded applications does, however, benefit from memory protection. The MPU hardware implements protection mechanisms without virtualization, which is then used in applications where this is important.

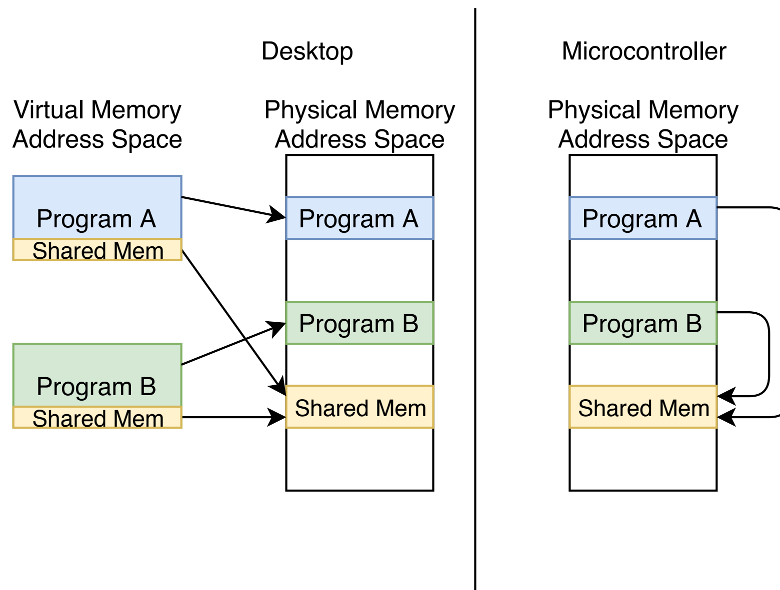


Figure 3.1.: Desktop vs. microcontroller memory systems

Figure 3.1 illustrates how the memory addressing is different in the two systems. The desktop system employs virtual memory and each of the programs reside in their own

virtual world, which is translated to the physical address space. The shared piece of memory is stored only once in the physical memory but mapped to each virtual memory space. In the microcontroller system both programs and data are located in the same address space and accesses the shared block using the same addresses.

Note that figure 3.1 does not show memory *protection*, which applies to both the left and right part of the illustration: In the desktop version it comes with the virtualization, while in the microcontroller version three MPU protection regions may be set for the memory blocks.

### 3.2.1. Microcontroller Shared Memory Problem Example

As previously discussed, the default shared memory architecture with selective restrictions applied in microcontroller applications has weaknesses due to the sharing. In this section, the memory protection scheme implemented in a known microcontroller software is discussed as a practical example. The software is the TMS570 Cortex-R port of FreeRTOS. Although the port applies to only this line of processors, the weaknesses discussed are relevant to all software designed for the Cortex-M and Cortex-R processors that also makes use of memory protection in a similar fashion.

### 3.2.2. MPU Setup

In the TMS570 Cortex-R port of FreeRTOS, the kernel functions are put in a special section called `kernelTEXT`, placed in the first 32 kilobytes of persistent flash memory. This makes it extremely easy to allocate an MPU region to protect it against illegal accesses: the code is aligned to the first 32 kilobytes and the size is a power of two. This region is set up this way regardless of how much memory is actually required. It may be tweaked by the application programmer for efficiency. When a new task is set up, regions are initialized as in table 3.2. Regions 0 - 3 and 11 are set once on startup as default regions, while regions 5 and 6 are set in context switches and are per-task specific.

The memory type refers to the kind of memory protected: flash for the internal persistent flash storage where the program and read-only data is stored, RAM for the volatile storage, peripheral for the microcontroller peripherals and system for internal system hardware such as clock generators.

The address and size is the absolute physical address in memory where the region is allocated.

The user, privileged and execute columns describe the access rights for the regions. User refers to the execution of software with the lowest execution privilege while privileged refers to privileged software such as the operating system. The execute column decides if the contents is executable or not and applies to both execution contexts.

Table 3.2.: Allocated memory protection regions

| Region number | Type | Address | Size | User | Privileged | Execute |
|---|---|---|---|---|---|---|
| `0` | Flash | `0x00000000` | 4MB | RO | RO | Yes |
| `1` | Flash | `0x00000000` | 32KB | NA | RO | Yes |
| `2` | RAM | `0x08000000` | 512KB | RO | RW | Yes |
| `3` | Peripheral | `0xF0000000` | 256MB | RW | RW | No |
| `5` | RAM | `0x08000000` | 512KB | RW | RW | Yes |
| `6` | RAM | `0x08000000` | 4KB | NA | RW | No |
| `11` | System | `0xFFF80000` | 512KB | RO | RW | No |

Legend: RO - Read Only, RW - Read and Write, NA - No Access

In short, a normally created task will have read/execute permissions to flash and read/write/execute permissions to Random Access Memory (RAM), excluding the regions allocated to the kernel. It will also have read/write access to all peripherals and read access to system modules, as well as all other threads and processes running on the system. Since the FreeRTOS kernel is extremely simple, access to all drivers, file systems etc. is also shared.

### 3.2.3. Shortcomings

There are some shortcomings with the memory protection approach used in the port. Most importantly, individual tasks are not separated from each other as they have exactly the same permissions. The most dangerous problems are read/write access to all peripherals by all tasks and each others memories. The problem of individual task separation is possible to address by some extent by removing region 3 from the table and using `xTaskCreateRestricted` for specifying region 5 and 6 manually. Unfortunately region 5 is reserved for stack usage leaving only one region to the user for either peripherals or normal memory. Overcoming this limitation would require modification of the context switch. By default, the MPU is not used to detect stack overflows.

### 3.2.4. Privilege Escalation

Since all FreeRTOS functions are located in the protected `kernelTEXT` section, precautions have to be taken in order to call them. This constraint applies to all FreeRTOS functions including the very basic delay function `vTaskDelay`. In order to perform such function calls, the privilege of the calling task is temporarily escalated by the use of a special system call called `prvRaisePrivilege`, permitting it to call into the protected `kernelTEXT` section. This system call sets the processor mode in the Saved Program Status Register and returns the old state, which is restored after calling into the protected section by using a separate system call. This is implemented as a macro called

`portRESET_PRIVILEGE` which internally calls `portSWITCH_TO_USER_MODE`. The code for the delay function is shown in listing 1.

```
1 void MPU_vTaskDelay( const TickType_t xTicksToDelay )
2 {
3     BaseType_t xRunningPrivileged = prvRaisePrivilege();
4     vTaskDelay( xTicksToDelay );
5     portRESET_PRIVILEGE( xRunningPrivileged );
6 }
```

Listing 1: The dangerous `vTaskDelay` function

This example code overrides the original `vTaskDelay` using a macro to implement privilege escalation before calling the actual `vTaskDelay`

The privilege escalation handler does not check that the caller is indeed allowed to do so. From a security perspective this is extremely bad as any code could call it and even escalate its privilege permanently. In the ideal state the protected functions would have to be called from within the service call handler and check that the operation is permitted before performing any actions. The current approach, although flawed, is still useful as it will help detect programming errors and failures when used properly since it can detect accidental memory violations but not deliberate ones. The use of privilege escalation instead of system calls relates to FreeRTOS not accepting calls from within Interrupt Service Routine (ISR)s, as it will not be able to context switch in that state.

Since the privilege functions may be called from anywhere, it is possible to use it in user code as shown in listing 2.

```
1 void DoEvil (void)
2 {
3     prvRaisePrivilege ();
4     ApplyFullCarThrottle ();
5 }
```

Listing 2: Privilege escalation function

If an attacker is able to inject code that calls into the `prvRaisePrivilege` function the attacker will have access to the entire system. This assumes that the location of the injected code is executable, which also is the case in the discussed port.

### 3.2.5. Summary

Denying access to other tasks memory is possible using the FreeRTOS Application Programmer Interface (API). All memory used by a specific task would have to be placed either on the stack or in a separate and aligned section using the GNU Compiler Collection (GCC) `__attribute__` mechanism. Region 6 would then be set to this section with region 5 for the stack. Tasks would then only have write permissions to their own memories. Using this approach would introduce additional requirements to the end application

programmer, forcing everyone to understand and use the MPU, a great disadvantage. Additionally every task would still have read and execute permissions to most memory locations.

Overcoming all of these obstacles would require all tasks to have their code and data in separate and properly aligned memory sections. An implementation of such separation in a completely custom OS by the use of process loading and relocation is discussed in chapter 5 and onwards.

## 3.3. Research Motivation

The Orbit student organization at NTNU intends to build a small CubeSat satellite and launch it into orbit. Its purpose will be to display images on a small screen and then take pictures of the screen with the earth in the background. Although this task may seem simple enough, robust and fault tolerant systems are needed due to space radiation effects causing spurious faults. Furthermore, the lack of physical access once launched requires a robust solution since loss of communication may cause total loss of the satellite.

In the heart of the satellite there is a computer responsible for the satellites vital functions. The computer must maintain radio contact with the ground operator, ensure correct orientation of the satellite, manage power consumption, data and commands. It must also handle any error conditions that may occur in any of the satellite systems. Failure in the processing of any of these functions may render the satellite inoperable. Due to the CubeSat form-factor limiting the size of the satellite to $10 \times 10 \times 20$ cm, there will be strict limits on power dissipation and redundancy in components.

These limits favours the use of traditional microcontrollers for the core computer components. Modern microcontrollers differ from their larger computer counterparts in their memory system architecture and speed, capacity and peripherals. Typically, a “large” microcontroller has 1 MiB of flash memory and 192 KiB of system RAM. Once put into a production environment the flash memory serves as read only storage for the application code and its resources. Devices sold in higher pin-count packages often have support for external memories allowing an expansion into the multi-megabyte range at the cost of complexity and speed.

The hardware limits often impose severe limitations on the software that runs on them. Microcontroller software is traditionally developed in a “bare metal” fashion, that is, there is no general purpose operating system. The program is instead developed as one monolithic application that interacts with hardware registers directly and is limited to only small and simple software libraries.

Developing critical software as one monolithic application introduces additional risks since tasks that would normally be loosely coupled or completely separate now have access to each others data. A bug in one software module may cause a fault or crash to appear in another, and it will be much harder to detect and correct such errors. In order to ease this problem, ARM Cortex based microcontrollers include a memory protection unit that, when enabled, is used to protect certain areas in memory from

unauthorized access. This unit may protect up to 8, 12 or 16 areas depending on the silicon implementation, and each area must have a size in the power of two, aligned to an address with the same power of two. The limitations of the MPU hardware combined with traditional monolithic application development results in the memory protection not being used at all, or only for partial separation of tasks. In the latter case only some of the possible access violations are detected and all violations are typically handled by restarting the entire application, causing data loss and interruptions in task execution.

In a system where strict task separation is desirable and crashes should only affect the module that causes it, more efficient use of the memory protection unit must be implemented. The main study of this paper is a protection scheme that fulfills these properties.

The end solution notably not limited to small satellite applications. It may be employed in any system that would benefit from stricter memory handling, thereby improving the characteristics of the product.

### 3.3.1. Previos Project Work

As part of the pre-thesis project work during spring 2018 research into fitting hardware for a small student-made satellite was made. The organization intended to build a small student-made satellite using low cost commercial off the self components. In order to integrate the various components of the satellite, robust software had to be written that runs on error-tolerant hardware suitable for deployment in space. The focus of the project work was

- to investigate the microcontroller market in order to find a device that is more suitable for the environment it will be exposed to in space, compared to the currently used device. As the market has evolved, better devices may be available. Important aspects of this evaluation are power consumption, error detection, error correction, general capabilities and general fault tolerance.
- to investigate the consequences of migrating to a new microcontroller.
- to make use of the Memory Protection Unit (MPU) available in most ARM-based microcontrollers as well as other safety features, in order to sandbox and separate software modules.
- to organize code, documentation and instructions, making continued development easier for newcomers.

The end result of the research was an evaluation of various hardware components which resulted in a new design of the on-board computer using the Texas Instruments TMS570LS1224 microcontroller. This is the hardware that the implementation presented in this thesis runs on and is intended for, however it also works with hardware designed by St Microelectronics. In theory it should be portable to almost any ARM Cortex-M or Cortex-R based microcontroller.

Typical for this hardware is the processor core manufactured by ARM and internal RAM sizes of 192 KiB. The TMS570LS1224 features an ARM Cortex-R4F processor at 168 MHz with floating point processing capability. The main reason why it was chosen over competitors is its extensive hardware safety features. It provides Error Correcting Code (ECC) for all of its main memories to detect physical corruption, parity on all peripheral memories, and most importantly two physical CPUs executing the same code in lock-step. What this means is that if a calculation error is made due to radiation it will be detected, so that software may perform the appropriate actions.

As part of the project the software ecosystems of the devices were evaluated. However, parts of the operating system presented in this thesis had already been developed. It was in the authors interest to continue with this development and implement the planned features, which would end with a system that is more convenient and safe than the existing solutions. The outline of this thesis was then suggested to the supervisors and work started autumn 2018.

# 4. Project Scope

This section lists the questions driving the solution research. They define the goals of the research, and in turn the expected output of this thesis. These questions make the goals during development as shown in chapters 5, 6 and 7. The solutions are tested in chapter 8 which gives a foundation for evaluation and discussion in later chapters.

## 4.1. Problem Summary

This section provides a simple summary of the existing challenges and provides an introduction to the solutions discussed in later chapters.

### 4.1.1. Memory Protection Schemes

The memory protection schemes used in desktop and microcontroller systems differ mainly in their use of virtualized memory, which is not available for microcontrollers due to its cost. Virtualized memory technology is however not really necessary for strict memory protection but it does make it easier to implement.

Commonly available microcontroller processors employ an MPU that enables up to 16 protection regions to be specified. Due to the lack of virtual memory and the low number of protection regions there are no known open-source implementations of a default non-shared memory scheme, since it is too complex to implement for the often "simple" applications. Instead, most applications have everything shared by default and use the MPU selectively, where each block of protected memory is manually set up by the programmer.

### 4.1.2. Key Difficulties

The most common reasons as to why a desktop-like approach is not used for microcontroller systems are summarized in the list:

1. Page misses generate non-deterministic performance in desktop systems. Many embedded designs require deterministic performance due to low-level hardware interaction, although this concerns only a small percentage of the code.
2. Live memory defragmentation is not possible, due to non-virtualized memory.
    - The problem is made worse by the power-of-two requirement of the MPU causing "pockets" of free memory that can not be remapped and used for larger sections. A memory allocation may fail despite there being enough

memory available, since the available memory is fragmented or not correctly aligned with the MPU requirements.

3. The number of entries in the MPU table is few, a true generalized approach would generate a lot of page misses

4. The high programmer effort of aligning data sections in monolithic applications. Each region requires individual programmer attention, both in setting the MPU table entries and assigning the correct data to the section.

5. It is often perceived that strict memory protection is not needed, despite clear advantages. Overconfident programmers think their code is safe and robust.

6. The majority of embedded applications does not require strict memory protection, making it a specialty feature.

### 4.1.3. Benefits

The important contribution presented in this thesis is the automated usage of the MPU for a by default non-shared memory architecture. This includes the necessary algorithm for allocating memory aligned to the requirements of the MPU, software for loading code using the algorithm, and a paging scheme for dynamic allocation and sharing.

The end result is a system with a sharing policy similar to the desktop counterpart: memory is by default not shared and any shared portions are explicitly agreed on. With the solution properly integrated into the operating system, the efforts required by the programmer is effectively zero.

Noticeable effects of the solution is increased security and reliability of the application. By keeping programs separate and making them unable to access each others memories, confidentiality, integrity and availability is ensured. By catching errors in a program and preventing it from crashing or restarting unrelated software, the perceived reliability is improved.

### 4.1.4. Hardware Popularity and Generalization

The solution applies to all ARM Cortex-M and Cortex-R based 32-bit microcontrollers with the MPU present. With some effort it may also be ported to the more powerful ARM Cortex-A processors, or even other core architectures.

In 2015, the microcontroller market grew to 5.5 billion devices with ARM devices having a market share of 25%. In the increasingly popular Internet of Things (IoT) market of wirelessly connected devices, ARM had a market share of 60% [16]. In 2016 a total of 17.7 billion ARM devices had been shipped [17]. In other words, the solution may help developers achieve better results in many cases, improving reliability, confidentiality, integrity and availability.

## 4.2. Research Questions

### 4.2.1. Question 1: How may memory contents be protected during various failure modes?

- RQ1.1: How may accidental programmer-induced errors be detected and handled? Programmer-induced errors cause both expected and unexpected failures. If these errors write to the memory of some unrelated program, that unrelated program may experience a failure itself. Preventing an error from escalating will improve reliability considerably, and prevent data loss.

- RQ1.2: How may intentional security breaches that attempt to read or even modify information belonging to a program be detected and handled?. Any successful attempt may reveal important confidential information, cause information corruption and/or a loss of service. Restricting the possible attack vectors will make it harder for the attacker to accomplish his/her goals and ideally make certain attacks impossible.

- RQ1.3: How may hardware failures be detected and mitigated? An unconfined hardware failure may cause errors similar to that of programmer made errors, compromising the integrity of the product. Examples of such errors are failed memory devices, physical damage, and radiation. Better handling of such failures may prevent a catastrophic failure in the field allowing the failing product to be replaced before any serious damages may occur.

- RQ1.4: How may enhancements to the system be made without adding developer effort? A system that is hard to use is unlikely to be used by most developers, or even worse, not be used properly, causing more and unexpected failures.

### 4.2.2. Question 2: How may the improvements be made without harming performance?

- RQ2.1: How may loading overhead be avoided? Loading is the process of starting the application, and added overhead is undesired for applications that are often used periodically.

- RQ2.2: How may execution overhead be avoided? Execution overhead is the time and performance lost to the implementation of the protection system at runtime, that is, while the application runs and is used.

A typical microcontroller application has deadlines to meet due to direct hardware interaction. For instance, when moving a robotic arm, the program must stop the motor at the correct time to achieve the correct degree of motion. If the deadlines are not met consistently the arm will appear unusable.

A solution that has too high a high performance penalty would be less competitive in the market and suffer from a low adaptation rate.

The engineering challenge is to achieve excellent protection mechanisms without affecting performance to the point where the application can not accomplish its task successfully.

Benchmarks and measurements demonstrating the effects of the implementation must be provided to present a good foundation for evaluation of the selected solution(s). If evaluation of the solution(s) is too time consuming to be performed in detail, the effects must instead be theoretically evaluated.

### 4.2.3. Summary

By implementing strong memory protection, errors will be detected much earlier and subsequently contained before escalation. The challenge is to overcome the hardware limitations of embedded microcontroller systems while still maintaining performance characteristics.

In order to arrive at a solution that will satisfy the above research questions, an understanding of existing techniques must first be made. The bulk of this discussion has already been performed in the Background chapter, chapter 2.

The solutions and implementation details themselves are discussed from chapter 5 and onwards. These chapters cover many details not outlined in the Background chapter, such as data structures and software APIs.

## 4.3. Solution Roadmap

In chapter 5, a memory allocation scheme is presented that enables easy allocation of memory aligned to the requirements of the MPU. Since this solution is generalized it will work on any Cortex-based microcontroller with small changes. The allocation algorithm enables protection aware allocation for the other sub-systems, making it possible to fulfill research question 1.1 to 1.3.

In chapter 6 the process of loading a program is discussed and presented. By presenting a tool that enables automated application of the memory protection, the first research drivers will be answered. By the use of a standard API, the tools will be easy to use, and question 1.4 will also be covered.

The issue that remains to be solved is dynamic memory allocation. Dynamic memory is memory that is requested after the program has started, and is usually used for situations where the programmer cannot know the required buffer sizes at compile time. A basic memory paging scheme is presented in chapter 7 that enables protection of dynamically allocated memory. Deterministic performance is solved by not swapping table entries for the fixed size text and data sections. Page misses, fragmentation and memory capacity issues can unfortunately not be solved due to the lack of virtualization technology which is a hardware limitation. The effect of page misses is discussed in section 8.3.1.

## 4.4. Architecture Overview

The solutions presented in the next three chapters are part of an operating system, which is much larger than the discussed parts alone. At the time of writing this section, the entire system consists of approximately 59,000 lines of code: 26,329 in the kernel alone, 5,127 in core utilities, 7,432 lines in supporting library code and 19,915 lines in regular programs written for the system. These are all original work.

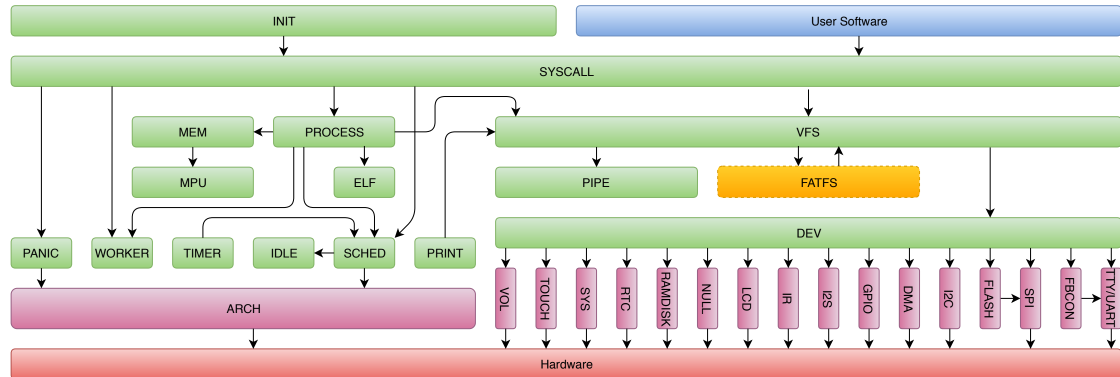


Figure 4.1.: Overview of the system architecture

The Operating System (OS) is a monolithic kernel, designed to be expandable through kernel modules and driver modules. Figure 4.1 provides a simplified illustration of how the most major components are put together. The green and purple components are the software modules that has been written for the project, while the orange components are third party modules. In this case there is only one third party module: the FATFS file system driver, which is statically linked to the kernel at compile time. The purple components cover the software modules that are architecture specific to a specific micro-controller, series or manufacturer. Green components are general purpose components, common to all configurations.

User software interacts with the system call module. This module provides the Application Binary Interface (ABI) which separates user-mode software and privileged kernel-mode software. Making use of the ARM processor Service Handler (SVC) instruction, user software does not have to know the address and implementation details of kernel-mode functions. A program compiled for one specific kernel version will also function on another as long as the ABI remains stable and the required functionality is included in the configuration.

The modules shown in the right-hand half of the illustration are related to the file system, file system drivers and device drivers. System calls and other modules interact with the virtual file system which provides an identical interface to all types of files. The OS currently provides three kinds of file systems. The most important file system is `dev`, providing access to device drivers through the `/dev` directory. The pipe file system provides the necessary backend for the pipe API: a means of inter-process communication. Both named and anonymous pipes are supported.

The modules shown in the left-hand half of the illustration are the more general core modules of the system. The dependencies shown are greatly simplified due to the clutter it would create in the figure if included. Important modules in this section are those that are relevant to process handling and loading, memory allocation, scheduling, error handling and timing. The arch module contains all the architecture specific code that is necessary for other modules to function, such as the context switcher, interrupt vector tables etc.

The parts that are related to chapter 5, Memory Allocation, is the green items labeled `MEM` and `MPU` in figure 4.1. It mainly interacts with a subsystem that helps it with MPU calculations, and is used by the process subsystem described in chapter 6. This process subsystem, responsible for loading programs, is represented by the `PROCESS` box. Due to its central importance, it interacts with many of the other systems presented in the figure. The pager presented in chapter 7 is part of the `SYSCALL` block, which interacts the the `PROCESS`, `MEM` and `MPU` subsystems.

# 5. Memory Allocation

In this chapter, the algorithm for allocation of memory with respect to the rules imposed by the MPU is presented. This algorithm is used in further chapters when programs are loaded, guaranteeing that each program cannot access each others memories. Communication is then made through protected operating system interfaces, such as pipes, files and device drivers.

## 5.1. Allocation Rules

Recall the rules from section 2.2.2 that the MPU hardware restricts the characteristics of an allocation. These are:

- The size of the protection region must be a power of two.
- The base address of the protection region must be aligned to that same power of two.
- There are eight sub-region disable bits that may optionally be used to restrict the protection.
- The maximum number of simultaneously protected regions are either 8, 12 or 16, depending on the silicon implementation.

With the limited number of simultaneously protected regions it is desirable to add another restriction: a single allocation may use no more than one region. This prevents the application from running out of regions at unexpected times when previous allocations used more than one set of protection registers without it being strictly necessary.

Also recall from section 2.2.2 that memory fragmentation can not be solved without the virtualized memory technology that the MPU lacks. This creates one issue not previously discussed: where is allocation metadata stored? The simplest of allocation algorithms typically store this metadata, which often contains the size of the allocation and a pointer to the next block, right next to the user data. Since the ultimate goal of using the MPU is to disallow access to critical data, and a corruption of the metadata may cause allocation subsystem failure, the metadata can not be stored inside the protected region.

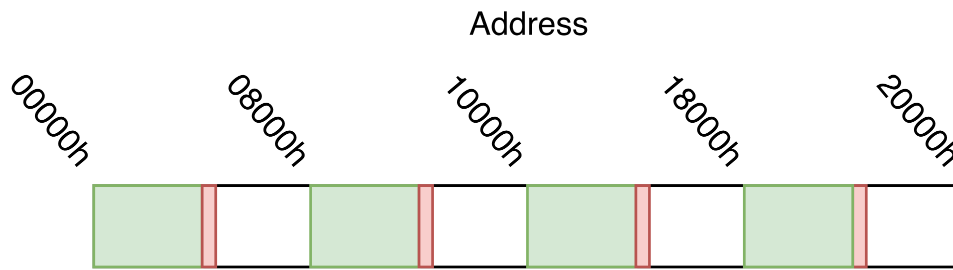


Figure 5.1.: Fragmented memory due to allocation metadata

The first alternative is to store the data right after the end of the protected region. This does however present a fragmentation issue: the base address of the aligned region that follows the one just allocated is unavailable for any new allocations, since the first eight bytes or so is occupied. This forces the allocator to skip a significant portion of memory to reach the next aligned address. Figure 5.1 illustrates the issue. A series of 16KiB allocations has been made, represented as green fill. The metadata has been stored at the end represented as red fill. In this case the red does not represent the actual scale of the metadata. Typically, the metadata size may be as low as 8 bytes or even less.

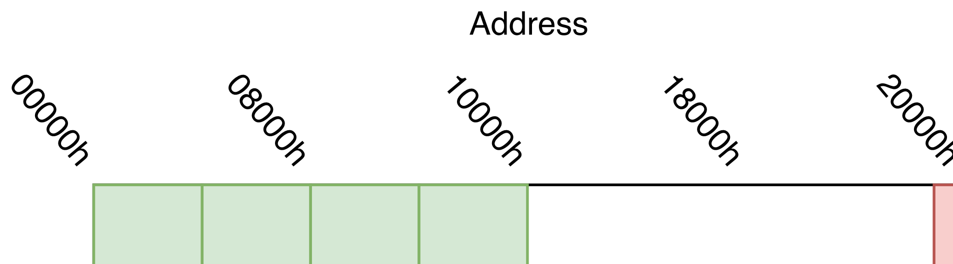


Figure 5.2.: Memory fragmentation avoided by moving metadata

As can be seen in figure 5.1 there are four allocated 16KiB blocks for a total of 64KiB. The total amount of free memory would then be $16KiB \cdot 4 - 8B \cdot 4 = 65504B$. However, since the memory is fragmented, no further 16KiB blocks may be allocated since all the aligned blocks are occupied! The solution is then to store all the metadata separately in its own block.

Figure 5.2 illustrates the solution, with the metadata moved. It is now possible to allocate three more 16KiB blocks without issue. This does, of course, not solve all fragmentation issues. Fragmentation caused by frequent allocation and de-allocation of different sized blocks may still cause fragmentation which can not be solved easily.

### 5.1.1. Allocation Rule Summary

The problems and restrictions described in the previous section is summarized into the following requirements for the allocation subsystem:

- The size of the protection region must be a power of two.
- The base address of the protection region must be aligned to that same power of two.
- There are eight sub-region disable bits that may optionally be used to restrict the protection.
- A single allocation must use no more than one protection section. This is due to the very limited number of hardware registers available.

## 5.2. Metadata Structures and Multibanking

Modern microcontrollers often contain more than one type of RAM for different purposes. For instance, the STM32F4 series contain 64KiB of Core-Coupled Memory (CCM) which has different characteristics compared to normal RAM. Another example is battery-backed RAM often connected or associated with a real-time clock, or "wall time" clock. In order to make allocation from different banks possible, the metadata structure has to be duplicated for each bank and function calls must take an argument specifying which bank to allocate from. This is kept in mind when designing the allocation structures and algorithm. The structure used once for each bank is listed in listing 3.

Next, each memory bank must keep track of the base address and size of the memory it allocates memory for. These are the `bankaddress` and `banksize` members of the structure.

In order to provide memory usage statistics, aid debugging and speed up certain operations it is necessary to provide metrics of the current amount of free memory, the lowest amount of free memory recorded and the end address of the memory bank. These are the `currentfree`, `lowestfree` and `lastaddr` member variables.

Next is the allocation table itself. A fixed allocation table is used due to simplicity and determinism in execution. The disadvantage of this scheme is that the list size may need to be tuned for optimal performance and memory usage.

The allocation table itself is made out of three variables. The first is the base address of the allocated block, `addr` in `MemTableEntry`. The most significant bit of this variable is reserved as a flag deciding if this memory address is in use or not. Strictly speaking this is non-portable but saves some memory by dropping yet another variable or bitfield for the purpose. The next variable, `nextaddr`, holds the address of the next memory block in the chain. The variable also doubles as the end-of-list designator if it equals the `lastaddr` member of the parent structure. At last there is the `memilist` variable. It serves a function similar to `lastaddr` by containing the *index* of the next table row in the chain. This member is set to the integer value of 255 for unused entries. Since the list is restricted to a maximum of 256 entries the `memilist` is an 8-bit variable and stored in its own array to prevent the compiler from adding padding or alignment space.

```
1 struct MemBank
2 {
3     unsigned int bankaddress;
4     unsigned int banksize;
5
6     unsigned int currentfree;
7     unsigned int lowestfree;
8     unsigned int lastaddr;
9
10    struct MemTableEntry memlist[MEM_LISTSIZE];
11    unsigned char memilist[MEM_LISTSIZE];
12
13    struct MemBank *nextbank;
14 };
15
16 struct MemTableEntry
17 {
18    unsigned int addr;
19    unsigned int nextaddr;
20 };
```

Listing 3: The memory bank structure

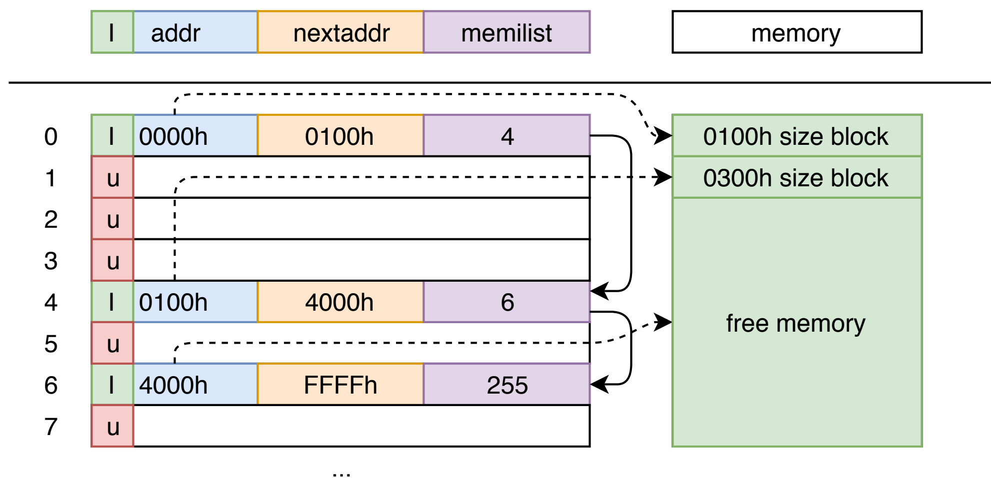


Figure 5.3.: Example of the memory list layout in memory

Figure 5.3 illustrates how the list appears in memory. In this case there are two blocks of memory allocated with the rest being free memory. As shown in figure 5.3 the table itself may be discontinuous and have free "slots" in-between used entries. When memory is allocated from the block of free memory, the free memory block entry is split using one of these free entries. With a trivial search for free entries the search will take linear time, however getting the next block in the chain or freeing a block will always take constant time.

What is not shown in figure 5.3 is where the metadata is actually located. In a system with multiple banks of memory, the data block does not even have to reside in the same memory. For most applications however the metadata will be stored in a reserved portion at the beginning of the memory with `bankaddress` and `banksize` referring to the rest.

## 5.3. Allocation Algorithm

In this section, the algorithm is first presented without fulfillment of the MPU requirements to aid understanding of the basis solution. This may be used to allocate memory used from within privileged code that does not use the MPU or as a general-purpose allocator.

A second version of the allocator is then presented with fulfillment of the MPU requirements. Since it uses the same metadata structures the two schemes are interchangeable. The choice of algorithm is then up to the operating system programmer, depending on whether the memory will be protected or not.

### 5.3.1. Algorithm Without MPU Support

#### Initialization

The memory allocator must first be initialized. This is done by initializing the `MemBank` structure. First, the entire table is cleared with `null` addresses and an `memilist` value of 255 indicating that there are no entries following it.

The very first entry is then set to indicate the entire memory as free. This is done by setting the address to the beginning of the memory and the next address to the end. As discussed previously the next address being equal to the end of memory also indicates the end of the table. The most significant bit of the memory address is cleared to indicate that the memory block is free.

#### Algorithm Outline

Algorithm 1 lists the top-level portion of the algorithm that takes the first function call from the requester. Only the desired bank and number of bytes are specified. It is then the task of this function to check the arguments for validity, fetch the correct bank metadata and find a free memory block large enough to satisfy the request. Once a block has been found the rest of the block is handed over to `MemAllocEntry` which is responsible for any table alterations and entry splitting.

```
Input   : nbytes ← Number of bytes to allocate
Input   : bankno ← The index of the bank to allocate from
Output: The address of the allocated block or zero on failure

if nbytes <= 0 then
  | return 0;
end

bank ← GetMetadata(bankno);
Align nbytes to the machine word size;
pos ← 0;

while MoreEntriesInList(bank,pos) do
  | current ← bank.memlist[pos];
  |
  | if InUse(current) then
  |   | if current.nextaddr == bank.lastaddr then
  |   |   | return 0;
  |   | end
  |   | pos ← bank.memilist[pos];
  |   | continue;
  | end
  |
  | blocksize ← (currenbt.nextaddr & MSB) - current.addr;
  |
  | if blocksize >= nbytes then
  |   | return MemAllocEntry(bank, current, blocksize, nbytes, pos)
  | end
  |
  | pos = bank.memilist[pos];
end
return 0;
```

**Algorithm 1:** Top-level algorithm outline: MemAlloc

Algorithm listing 2 specifies the operation of the entry allocator. Its job is rather simple: split the block if necessary, set the block as in use, update memory consumption statistics and return. Note that if the splitting fails due to a lack of free table entries, the `MemAllocEntry` algorithm will fail with zero which in turn makes the entire allocation fail. This prevents an almost out of memory condition from consuming *all* the remaining memory by failing early.

**Input** : bank ← The bank to allocate from
**Input** : current ← The table entry to consider
**Input** : blocksize ← The size of the entry's memory block
**Input** : nbytes ← Number of bytes to allocate
**Input** : pos ← Position of the current entry in the table
**Output:** The address of the allocated block or zero on failure

address ← current.addr;

**if** *blocksize >= nbytes* **then**
| SplitMemoryBlock(current);
**end**

current.addr ← current.addr | MEM_INUSEBIT;
UpdateStatistics (bank);
**return** *address*;

**Algorithm 2:** Entry allocator: MemAllocEntry

### Deallocation

Deallocation of a memory block is the same for both MPU-less and MPU-enabled memory. This procedure is shown in algorithm 3. It finds the table entry for the allocation, marks it as free and merges the now free memory with the surrounding blocks in the list. By checking and merging both backwards and forwards in the table, free memory blocks are always represented as one continuous block entry in the table. Memory consumption statistics are updated to reflect the new memory usage.

**Input** : address ← The address of the memory to allocate
**Output:** Non-zero for failure or zero on success

bank ← FindBankForAddress(address);

**if** *IsInvalid(bank)* **then**
| **return** *1*;
**end**

current ← FindEntryForAddress(bank,address);
current.addr ← current.addr & ~MEM_INUSEBIT;
UpdateStatistics (bank);
Merge with the next table entry if possible;
Merge with the previous table entry if possible;
**return** *0*;

**Algorithm 3:** Memory deallocator: MemFree

With the base allocation algorithm clear, it is time to move on to the MPU enabled version.

### 5.3.2. Algorithm With MPU Support

#### MPU Register Summary

In order to understand the optimization of the MPU-enabled allocator it is benificial to know and leverage the structure of its registers. The register layout of two registers in the Cortex-M Central Processing Unit (CPU) is used: the Region Base Address Register (RBAR) and the Region Attribute and Size Register (RASR) [9, 10]. The fields important for allocation are illustrated in figure 5.4 and 5.5.

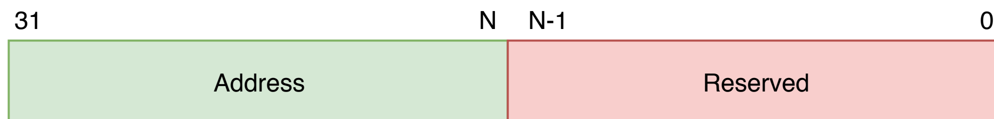


Figure 5.4.: Simplified Region Base Address Register

The number of bits to use in the RBAR depends on the size of the region. Each power of two size increase of the region adds a bit to the leftmost field. This field can have a maximum of 27 bits corresponding to a 32-byte region.

The RASR contains three relevant fields: the Sub-Region Disable bits (SRD), size of the region and an enable bit. The encoding of the size field is Region size in bytes = $2^{\text{SIZE}+1}$. Note that region sizes of 128 bytes or less do not support subregions, so the SRD field must be cleared in those cases.

The reserved fields contain data not relevant to the allocation subsystem as well as actual hardware reserved bits. These reserved bits are to be set by the caller as they contain important access right attributes for the memory which the caller knows how to set.

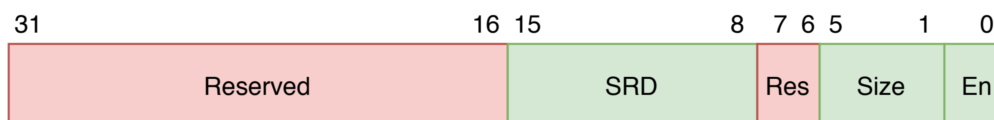


Figure 5.5.: Simplified Region Attribute and Size Register

Listing 4 describes the structure that the MPU-enabled algorithm fills with information. The `dataptr` member is the data pointer that must be used for accessing the actual memory while the normally returned address may be used for deallocation only. The distinction occurs because the protected memory allocated may have a different offset than the entire allocation due to alignment.

```
1 struct MemMpuReturn
2 {
3     unsigned int dataptr;
4     unsigned int mpu_rbar;
5     unsigned int mpu_rasr;
6 };
```

Listing 4: The structure used for MPU allocations

### Algorithm Outline

Algorithm 4 is the MPU-enabled variant of algorithm 1. It follows the same structure, with all the required code added.

As with the base version it starts with checking the input parameters and getting the metadata structure for the specified bank. The allocation size is then clamped to 32 bytes at the lower end. This is the minimum region size that the MPU hardware is able to protect so any allocations smaller than this must be clamped. The next step is to find the smallest region size that may be used, which is the specified number of bytes clamped upwards to the nearest power of two. This exponent is then stored in `mpuminimal` and its size in bytes counterpart is stored in `regionsize`.

Next up is accounting for subregions. Since the hardware only supports subregions for region sizes larger than 128 bytes a check must be made so that the allocation size is not adjusted for sizes that are too small. A copy of `regionsize` called `minalloc` is first made, which will be the smallest allocation that must be made. For small regions this will simply remain the same as the region size. For other cases it is instead set to the smallest multiple of the subregion size, using the `subreg` variable that was just set to one eight of the total region size. Before starting the main loop two more values are calculated: the number of spare subregions and the bitmask the region must be aligned to. The number of spare subregions gives the leeway that the allocator may experiment with when trying to fit the allocation inside a region and is stored in the `nspare` variable. The bitmask will after a bitwise and-operation with an address result in the base address of the corresponding region.

The main loop searches for free memory blocks just like the base version does. It gets the entry at the current position and checks if it is in use or not. If it is, the loop skips to the next entry or returns failure if there are no more blocks to check.

Next comes the key differences. First the memory size of the current entry is fetched along with its address and corresponding protection base address. The previously discussed bitmask is used to produce this address. The size of the block is then checked against the minimum allocation size to check if it will fit at all. If it does not then the main loop will skip to the next entry.

If the size does match then the next task is to see if it is possible to fit the minimum allocation size in the memory block given the calculated region size and subregion size. Spare subregions are added to the masked address `regionaddr` until the address of the to be allocated block is past the address of the free memory block, or it runs out of spare “leeway” subregions. A check is made to see if there was not enough spare subregions

to make it fit. In those cases the allocator retries with the next full MPU region by incrementing the `regionaddr` by `regionsize` and adding the gap to the allocation size. In reality there are more ways to approach this problems such as checking larger region sizes instead of just the next block, and to search for more fitting free memory blocks all together. This algorithm does however only run the first approach.

If the address checks out the alignment space is added to the size of the allocation. This memory will go unused, but cannot be avoided easily since the alignment is required. Another check is then made on the size to guarantee that the now larger allocation still fits. Free space at the end is split into a new entry for use later.

The allocation is now ready to be committed. The MPU registers are first calculated, here simplified by the `CalculateRegisterSettings` function. This function sets the RBAR and RASR register values that are used to initialize the MPU: base address, subregion disable bits and the size of the region. The `dataptr` member is also set, and will be the address of the region base or any of the subregion depending on how the spare subregions were used to move things around.

At last the in-use bit is set to indicate that the memory is now occupied, the memory consumption statistics are updated and the memory is cleared. Clearing the memory is important to prevent any program from obtaining information from a previously used memory location.

Note that there are three addresses provided from the algorithm:

- The address programmed into the MPU register.
- The address where the protected data starts.
- The address of the memory block that is later used when freeing memory.

**Input :** nbytes ← Number of bytes to allocate
**Input :** bankno ← The index of the bank to allocate from
**Input :** mpu ← Pointer to caller-supplied `MemMpuReturn` structure
**Output:** The address of the allocated block or zero on failure

**if** *nbytes <= 0* **then**
  **return** *0;*
**end**

Align nbytes to the machine word size;
Clamp minimum allocation size to 32;

bank ← GetMetadata(bankno);
mpuminimal ← Nearest power of two that will fit nbytes;
regionsize ← 1 « mpuminimal;
minalloc ← regionsize;
subreg ← regionsize / 8;

**if** *minalloc > 128* **then**
  minalloc ← Round to nearest multiple of subregion size;
**end**

nspare ← (regionsize - minalloc) / subreg;
mask ← ~(regionsize - 1);
pos ← 0;

*The algorithm continues on the next page.*

```
while MoreEntriesInList(bank,pos) do
    current ← bank.memlist[pos];
    if InUse(current) then
        if current.nextaddr == bank.lastaddr then
            return 0;
        end
        pos ← bank.memilist[pos];
        continue;
    end
    blocksize ← (currenbt.nextaddr & MSB) - current.addr;
    addr ← current.addr;
    regionaddr ← addr & mask;
    redo:
    if blocksize >= minalloc then
        addrc ← Add subregions to regionaddr until we are past the region block
          start;
        if addrc < addr then
            regionaddr += regionsize;
            minalloc += regionaddr - addr;
            goto redo;
        end
        minalloc += addrc - regionaddr;
        if blocksize < minalloc then
            goto redo;
        end
        if blocksize >= minalloc then
            if !MemSplit(b, current, minalloc, pos) then
                return 0;
            end
        end
        mpu ← CalculateRegisterSettings(addrc, regionaddr, subreg, nspare,
          mpuminimal);
        SetInUseBit (current);
        UpdateStatistics (bank, nbytes);
        ClearMemory (addr, minalloc);
        return addr;
    end
    pos = bank.memilist[pos];
end
return 0;
```

**Algorithm 4:** Top-level algorithm outline: MemAllocMPU

# 6. Program Loading

In order to make use of the allocator and the protected memory regions software must be written for it. This chapter presents a solution that does not require any extra programmer attention: each program is compiled separately and loaded correctly by the operating system during startup or at the request of a user command.

Instead of having the programmer set the protection mechanisms manually as in competing solutions, the allocator is used when loading code from a file. This enables the operating system to perform all alignment tasks without intervention.

## 6.1. Program Structure

The structure of a program must first be understood before the process of loading it can be presented. In order to use existing open-source development tools and thereby avoiding having to learn new systems, the multiplatform ELF file format is used. This format is described in greater detail in section 6.2.

Programs compiled with the open-source GNU toolchain are typically divided into four sections:

- `.text` is used for the program instructions. This section is typically given read-only and executable access rights. As long as this section is the only executable section, the program will not be able to modify itself and an attacker will not be able to execute arbitrary code.
- `.rodata` is used for read-only constant data. String constants are typically found here as well as other compiled-in resources.
- `.data` contains pre-initialized read-write data. Its purpose is virtually the same as the `.rodata` section except that it may be overwritten during execution.
- `.bss` describes the normal working memory of the program. This section does not actually contain any data since the section is to be zeroed out at program start. Instead it simply specifies how many bytes to allocate.

Additional data sections are required for a program to function:

- The stack contains data temporarily used by the running program. Data stored here are typically return addresses for function calls and local variables. The stack grows and shrinks within certain limits during program execution.

- The heap contains data that is dynamically allocated by the program using the `malloc` and `free` library functions. These functions collaborate with the operating system to get free blocks of memory at runtime.
- The program arguments, which are given to the program as arguments to the main function. These arguments are often associated with program usage on the command line, used to let the user specify what the program is to do and with what files. In the C programming language the arguments are provided as the `argc` and `argv` variables to the main function.
- Environment variables given to the program at startup. The most important environment variable is typically the `PATH` variable which specifies where the shell may look for other programs, preventing the user from having to type out the full path to a program when starting it.

These sections may be grouped by access rights and intended usage for easier allocation and memory management:

- Executable read-only data, very high performance: `.text`
- Non-executable read-only data, medium performance: `.rodata`
- Non-executable read-write data, medium performance: `.data`, `.bss`, program arguments and environment variables.
- Non-executable read-write data, high performance: The program stack.
- Non-executable read-write data, medium performance, varying size: The program heap.

The performance distinction is made due to how latency affects the typical program. Many microcontrollers have multiple internal memories with different characteristics, which becomes important during allocation. Since an instruction fetch has to be made for every instruction, the `.text` section should naturally be allocated in the highest performing executable memory. With most functions requiring pushing and popping variables to the stack, the stack comes in at second place performance wise. All other sections will typically not be affected as much by latency and can be placed in external memories.

The effects of latency does of course depend on the program being run. A program that frequently references memory in the `.bss` section will be slowed to a greater degree than one which runs a compact algorithm using the CPU registers and stack only, for instance.

## 6.2. The ELF file format

The Executable and Linkable Format (ELF) file format is the format used by the open-source GNU is Not Unix (GNU) compiler toolchain and the Linux operating system,

among many other tools and operating systems. This file format is simple, yet at the same time able to work with many different machine architectures and also store arbitrary data. The modularity comes from the ability to store arbitrary named sections in the file. It is then up to the program interacting with the file to determine what a section is used for depending on its name.

```
1 struct ElfHeader
2 {
3     char magic[4];           // 0x7F + "ELF"
4     char versionpart[5+7+8]; // Members that are only checked for "sameness"
5     unsigned int entrypoint; // Program entry
6     ...
7     unsigned int shoff;      // Start of the section header table
8     ...
9     unsigned short shnum;    // Number of section headers
10    unsigned short shstrndx; // Section header table entry for the names
11 };
12
13 struct ElfSection
14 {
15    unsigned int nameoffset; // Index into shstrtab
16    ...
17    unsigned int offset;     // Offset in the elf file
18    unsigned int size;       // Size in the elf file
19    ...
20 };
```

Listing 5: The structures of the ELF file format

Listing 5 shows the relevant members of the two ELF file format structures with irrelevant data shown as three dots. First is the file header, always located at the very beginning of the file. The magic bytes are used to identify the file format in systems that does not rely on the file extension for identification. Next is the version information identifying the type of CPU this program is designed for among others. This information is only checked for match with a pre-defined array in the operating system loader which corresponds to the CPU architecture it runs on. The entry point defines the address of the first instruction in the `.text` section that is to be executed when starting the program.

The next three entries describe a list of section headers. Each of these sections specify an offset and size within the file where the actual data associated with it is stored. Note how the names of the sections are stored: the main header specifies the index of the section containing all the strings with the names, and each section header contains an index into this section.

```
$ arm-none-eabi-readelf -S sh.elf
There are 11 section headers, starting at offset 0x68b0:

Section Headers:
  [Nr] Name          Type        Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]               NULL        00000000 000000 000000 00      0   0  0
  [ 1] .text         PROGBITS    00000000 000034 00262c 00  AX  0   0  4
  [ 2] .rel.text     REL         00000000 005918 000cb8 08   I  9   1  4
  [ 3] .rodata       PROGBITS    0000262c 002660 000af4 00   A  0   0  4
  [ 4] .rel.rodata   REL         00000000 0065d0 000278 08   I  9   3  4
  [ 5] .data         PROGBITS    00003120 003158 00042c 00  WA  0   0  8
  [ 6] .rel.data     REL         00000000 006848 000028 08   I  9   5  4
  [ 7] .bss          NOBITS      0000354c 003584 0010fc 00  WA  0   0  4
  [ 8] .shstrtab     STRTAB      00000000 006870 000040 00      0   0  1
  [ 9] .symtab       SYMTAB      00000000 003584 001bf0 10     10 291  4
  [10] .strtab       STRTAB      00000000 005174 0007a4 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  y (noread), p (processor specific)
```

Listing 6: Example structure of a program

Listing 6 shows the section of a real program that may be loaded by the operating system running in a microcontroller. Note the familiar section names and their `.rel` counterparts. These contain relocation information which is discussed in section 6.3.

## 6.3. Relocation

Recall that microcontrollers lack virtualized memory technology and must use the real physical addresses at all times, which is one of the defining characteristics of the microcontroller. This creates two challenges that must be solved in order to load a program: a program stored in a file can not know where its data will be located, and when running two instances of the same program they must reference different memory locations.

The solution is a technique called relocation. A list of every location in the program that depends on the physical location of data is made at compile-time. When the program is loaded the data is first copied into freshly allocated memory. The list is then iterated upon, modifying every instruction and data reference to reflect the actual physical location of the data.

As can be seen in listing 6, each section containing data that must be modified has a corresponding relocation section containing relocation entries. Each entry contains three

pieces of information:

- The offset of the instruction or data that must be modified, relative to the corresponding section.
- The type of relocation to perform. Different instructions have different encodings which will require different relocation procedures.
- An index into the symbol table, `.symtab`. This table contains a list of all the symbols in the program such as functions and variables. The section and offset of the referenced function or variable are store here.

The section and offset located in the symbol table is used to get the real address of the target symbol by adding the allocation address of the section and offset. This address is then combined with the instruction or data entry that must be modified so that it references the actual location.

In order to load code on a Cortex-M3 or Cortex-M4 platform, three types of relocation must be supported:

- `R_ARM_ABS32`: Plain 32-bit memory references.
- `R_ARM_THM_CALL`: Call instructions.
- `R_ARM_THM_JUMP24`: Jump instructions. These relocations are the same as the call instructions.

For the Cortex-R two additional types has to be supported:

- `R_ARM_THM_MOVW_ABS_NC`: Move data into the lower portion of a register.
- `R_ARM_THM_MOVT_ABS`: Move data into the upper portion of a register.

The relocation types are explained in further detail in the *ELF for the ARM Architecture* [18] manual. It details each type of relocation and how to perform them. Combined with the instruction encoding information which are documented in the *ARM Architecture Reference Manual* [19], it is possible to develop a capable system for run-time code relocation necessary for dynamic loading.

## 6.4. Load API Selection

In order to let a programmer run programs when needed an API must be specified. By using an existing API specification and not designing an entirely new one, programmers may use existing knowledge and code, greatly simplifying the task and easing the job of the developer. The selection of the API relates heavily to research question RQ1.4.

In the POSIX-world there are two methods used when starting a program: fork & exec, and `posix_spawn`. These two approaches accomplishes the same task using different methods.

The first approach splits the creation of a process in two parts. First, the current process is said to be “forked”. It creates an exact duplicate of the calling process and both will continue execution from the same location in the code. The return value of the call determines if code is running in the child or parent process. At this point the environment of the child process may be modified greatly by user code such as closing or opening files, changing input/output devices and so on. At last the program code of the child process is replaced by the intended program using the `exec` call.

The obvious advantage of this approach is the ability to modify the running environment of the child process in virtually any way possible. What is not as obvious is the complexity of the fork. Desktop systems greatly enhance the operation by using the MMU for a copy-on-write scheme often avoiding the copy altogether. Embedded systems without the full MMU cannot do this and must instead do the entire process loading operation once more at the cost of both time and memory. The POSIX standard specifies an alternative API: `posix_spawn`.

`posix_spawn` attempts to fulfill the majority of fork and exec use cases. Listing 7 shows the function prototype. As can be seen, it takes the path of the program to execute, the familiar program arguments and environment variables. The Process ID (PID) is returned to the caller using a pointer while the function itself returns a status code.

```
1 int posix_spawn (pid_t *restrict pid, const char *restrict path,
2         const posix_spawn_file_actions_t *file_actions,
3         const posix_spawnattr_t *restrict attrp,
4         char *const argv[restrict], char *const envp[restrict]);
```

Listing 7: The `posix_spawn` function prototype

What is perhaps not as familiar is the `posix_spawn_file_actions_t` and `posix_spawnattr_t` arguments. As noted earlier a program may want to change the set of open files and/or change other state information. `posix_spawn_file_actions_t` provides the file functionality allowing files to be closed, opened and duplicated. `posix_spawnattr_t` enables setting of various attributes, such as the process group of the child process and its scheduling policy among others.

The `posix_spawn` technique is selected as it is the most suitable for microcontroller operation. Due to its place in the POSIX standard it is also available for other platforms such as Linux, keeping the ability to develop cross-platform code intact.

## 6.5. Load Procedure

The program load procedure is initiated by a `posix_spawn` system call from either a user program or the kernel. Typically a call from kernel space only happens when starting the very first process at startup with any further calls coming from user code.

```
Input  : cur ← The process data of the calling process
Input  : pid ← Pointer to an integer which will store the pid
Input  : name ← String containing the name of the program file
Input  : argv ← List of program arguments
Input  : env ← List of program environment variables
Output: Success state

argvcount, argvsize ← ProcessSpawnCountArgs (argv);
envcount, envsize ← ProcessSpawnCountArgs (env);

slot ← ProcessAllocate;
if !slot then
|   return 0;
end

CopyCurrentDirectory (slot, cur);
loader ← NewProcessLoader;
loader.fd ← open (name);

if loader.fd < 0 then
|   ProcessFree (slot);
|   return loader.fd;
end

ret ← ProcessSpawnLoad (slot, loader, argvsize, envsize);
close (loader.fd);

if ret then
|   ProcessSpawnFreeAllocatedMemory (slot);
|   ProcessFree (slot);
|   return ENOEXEC;
end

ProcessSpawnCopyArgs (slot.argv, argv, argvcount);
ProcessSpawnCopyArgs (slot.environ, env, envcount);
SetStartParameters (argvcount, argv, environ);

slot.ownerpid ← cur.pid;
slot.pid ← currentmaxpid++;
*pid ← slot.pid;

ProcessSpawnAttr (cur, slot, arg);
ProcessSpawnFileActions (cur, slot, arg);

SchedAddThread (slot, loader.entrypoint);
return 0;
```

**Algorithm 5:** Simplified process loading procedure

Algorithm 5 presents the procedure in simplified form. It starts off with enumerating the program arguments and environment, also calculating the storage space required. A new control structure for the new process is then allocated. This structure contain

important run-time data that will be used during execution by both the operating system and user code. Examples are the PID, the parent process PID, memory pointers, execution context, current directory, open file handles and so on. The current working directory is then copied into this new slot and a loader object is allocated for use by the ELF file format interpreter. This loader object contains temporary data and is discarded once the program is loaded. Next, the file containing the program is opened and the actual loading commences. This is greatly simplified in the pseudocode, in reality the `ProcessSpawnLoad` call is responsible for

- Interpreting the headers of the ELF file.
- Allocating the necessary memory blocks using the allocator from section 5.3.2. The structure and access rights described at the end of 6.1 is used.
- Loading the instructions and program data from the file into memory, and zeroing out the `.bss` and stack memories.
- Running the relocation process that makes all memory references correct in relation to the locations in which they were loaded to.

Once the program instructions and data is loaded into memory, the program arguments and environment variables are also copied to the freshly allocated memory. The program now has its own copy of this data that only it has access to. In order to be able to call its main function when execution starts the start parameters are set up with the scheduler. It is then given a PID and parent PID. At last the spawn attributes and file actions are processed before the thread of the program is added to the scheduler and the spawn implementation returns.

## 6.6. Context Switching

In order to keep the access rights to memory consistent with multiple programs running the context switch must swap the entries in the MPU hardware. This happens in addition to the swapping of base context registers. In order to modify the context switch the location of the MPU register data that will be loaded must be considered. This is best stored in the `ThreadControl` structure that contains scheduling data, shown in listing 8. The process of restoring the MPU registers are then shown in listing 9.

```
1 struct ThreadControl
2 {
3     // Pointer to current top of stack, must be first
4     unsigned int *stacktop;
5 
6     // Privileged or not, set to 1 for unprivileged, must be second
7     unsigned int privileged;
8 
9     // MPU data, up to 4 regions, must be third
10    unsigned int mpuregs[MPU_REGS_NEEDED];
11 
12    ...
13 };
```

Listing 8: The beginning of the `ThreadControl` structure

Listing 9 only shows how the MPU regions are set right after the scheduling algorithm has chosen what program to run. `currentthc` is a global variable that points to the `ThreadControl` structure of the currently running program that must now be restored. The pointer is first loaded and the `privileged` member is fetched from memory. This is used to set the processor state allowing certain kernel threads to remain privileged in what is otherwise the least privileged context level. For all normal programs, the privilege level will be unprivileged. Next the address of the RBAR is loaded and `r1` is set to the address of `mpuregs` in `ThreadControl`. The content if this array is then copied verbatim into the hardware registers as efficiently as possible using a multi-word load and store as long as the program runs unprivileged.

Not shown is how the context of the previously running program is stored, how a new program is selected by the scheduler, and how the rest of the new programs context is restored. Note that the illustrated code applies to the Cortex-M variant. Cortex-R code is slightly different as it lacks the memory mapped registers and uses special instructions instead.

```
1 ldr r3, =currentthc     // r0 becomes the stacktop member of currentthc
2 ldr r1, [r3]
3 ldr r0, [r1]
4 
5 ldr r2, [r1, #4]        // Use correct privilege level
6 msr control, r2
7 
8 ldr r3, =0xe000ed9c     // Get address of the MPU_RBAR register
9 add r1, r1, #8          // Point r1 to our data in ThreadControl
10 tst r2, #0x01          // If unprivileged, set mpu regs
11 itt ne
12 ldmne r1, {r4-r11}     // Copy using the alias registers
13 stmne r3, {r4-r11}
```

Listing 9: Restoring the MPU registers in assembly

# 7. Adding Paging

In this chapter, the implementation of a paging scheme that works without virtualized memory technology, using a microcontroller MPU is discussed. This solves the final problem of memory management: allocating more memory dynamically at run-time.

## 7.1. Paging without Virtualization

In previous chapters a method of loading a program with the full protection of the MPU was presented. There is one issue that has not been addressed yet: programs that does not know how much memory they will need at startup and naturally allocates and deallocates memory at runtime. Desktop systems naturally support this using the memory virtualization hardware which enables hundreds of pages to be managed with ease, and allocation issues to be solved.

Microcontrollers with MPU hardware will typically have a few free regions that may be used for "extra memory". Virtualization will of course not be supported so fragmentation issues can and will occur, however these extra memory regions may just be enough to get some programs working without too many issues.

## 7.2. Adding Paging

The task of adding paging can be divided into smaller tasks:

- Increasing space in control structures to accommodate for additional register values that must be swapped during a context switch.
- Adding a list of pages in the control structures. This list is used to resolve page faults. Ideally it must be able to grow and shrink as the program allocates and frees pages. A base solution may use a statically allocated list limiting the number of pages to some reasonable limit.
- Adding a new system call for allocating and freeing pages.
- Writing a page fault handler. This handler is responsible for swapping MPU protection regions in and out of the hardware registers when an exception occurs and then restart the faulted instruction.
- Optionally adding a way to set up shared memory regions across processes. Shared memory may then be used for high-performance communication.

### 7.2.1. Architectural Considerations

Desktop systems such as those using the `x86` architecture has fixed size pages of 4KiB each. This means that for a 32-bit architecture, 12 bits are used to address page contents while 20 bits address the page itself. Since the page size is fixed, the number of bits are also fixed and can be structured in a tree.

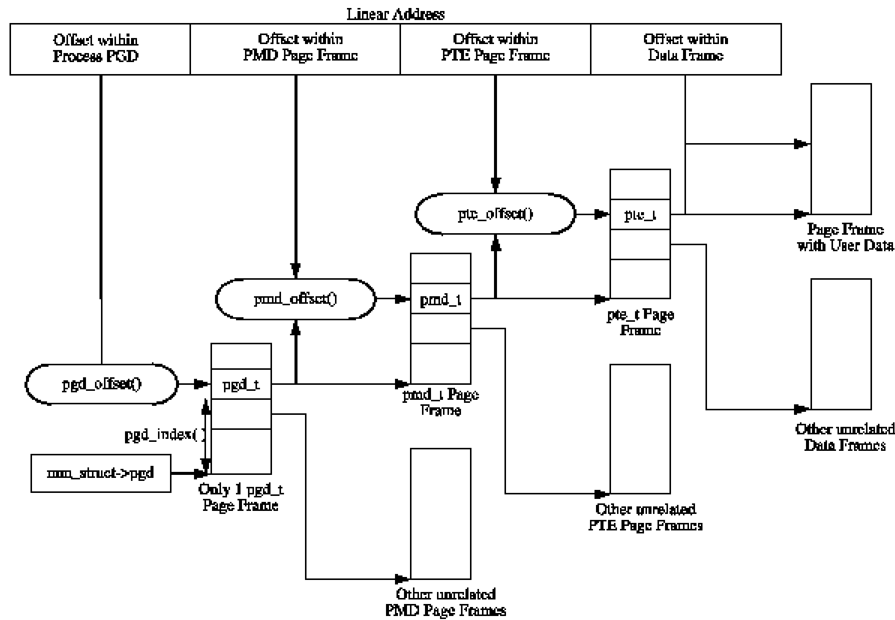


Figure 7.1.: Linux Page Table Layout[1]

Popular operating systems such as Linux does exactly this[1]. The page table is split into three levels: the top-level Page Global Directory (PGD), the next level Page Middle Directory (PMD), and the last level Page Table Entries (PTE). Each of these levels are assigned to a set of bits in the virtual address so that the most significant bits address an entry in the PGD, the next in the PMD and so on. The end result is the tree structure shown in figure 7.1. Since a typical process has many holes in its address space the page table may effectively be pruned.

As previously discussed the MPU hardware in the Cortex processors can protect regions that have a size of almost any power of two with data shifted within using the subregion disable bits. Although this enables protection of large amounts of memory with very few entries, it makes the management of the page table more difficult. Searching the table becomes even more difficult when taking the subregion disable bits into account since the start address of a region may no longer be the actual start address. With variable amount of addressing bits per page, a fixed tree cannot be used, at least not easily.

The most straightforward solution is to have a simple table of valid start and end addresses per region and then linearly search this table when a fault occurs. Although possibly very slow indeed, the flexibility of the region sizes enables for much smaller

tables. As long as the previously discussed regions for instructions and data are still fixed in the MPU registers the only accesses subject to page faults are those to memories explicitly allocated after program start. Those that does not make use of such "extra" memory allocation will still enjoy fully predictable execution with no page faults.

Since the primary focus of this thesis is not performance but the memory protection techniques themselves, this solution is acceptable as a base. Any further discussion will use this scheme applied for the Cortex-R processor.

### 7.2.2. Control Structures

In order to accommodate for the pager some control structures must be altered and some added. First and foremost the `ThreadControl` structure which holds the MPU register values to set on a context switch must be changed, since there are now six regions in use: two for paged memory and four for the old instruction and data sections.

The choice of two regions for paged memory is simply that two is the least needed to copy between two paged regions without frequent page faults while still maintaining simplicity. The replacement policy is also simple as will be discussed in section 7.2.6.

The number of regions stored in the `ThreadControl` structure, shown in listing 8, is controlled by the configuration variable `MPU_REGS_NEEDED`. The variable is changed from $3 \cdot 4$ to $3 \cdot 6$. Note that the Cortex-R uses three registers per region instead of two which the Cortex-M uses.

Next up is defining the new control structures for the pager itself. A list of pages must be created that is searchable by the current program PID and fault address. For a successful search the values which will be programmed into the MPU registers must be produced. Additionally the table may store auxiliary information necessary for managing the table. In order to shrink or grow the table there must be a way to append and remove data. This is achieved by organizing the table into blocks that may be appended to each other in a linked list.

```
1 struct PageTableEntry
2 {
3     unsigned int tid;       // Thread ID (same as PID for now), -1 = unused
4     unsigned int useptr;    // Pointer where the data is (accounting for subregions)
5     unsigned int maxptr;    // Highest address allowable + 1
6     unsigned int mpu_rbar;  // Region base address register
7     unsigned int mpu_rasr;  // Region attribute and size register
8     unsigned int freeptr;   // Pointer to use when freeing memory
9 };
10
11 struct PageTable
12 {
13     struct PageTableEntry entries[PAGER_MAX_ENTRIES];
14     struct PageTable *next;
15 };
```

Listing 10: The page table structures

Listing 10 shows the resulting control structure. The first three variables in each entry is used during search. Instead of searching for an address using a bitmask the address is compared to a minimum and maximum value, `useptr` and `maxptr`. This prevents the page walk algorithm from having to inspect the subregion disable bits in the RASR during search making the operation much faster.

The `mpu_rbar` and `mpu_rasr` values are programmed into the MPU registers on a match. As noted earlier the Cortex-R uses three registers, here the region attributes and region size data is combined into one reducing the size of the page table. These are split again when set in the MPU.

At last the entry contains a separate value required when a memory block is freed. For better cache locality, this entry may be split into a separate data structure, however with most microcontrollers lacking cache this is not an urgent issue.

The page table itself is made up of blocks of `PAGER_MAX_ENTRIES` each, and each block contains a `next` pointer for chaining. With each entry currently occupying $6 \cdot 4 = 24$ bytes the block size is a low 16 entries, chosen due to the low usage of dynamic memory in most microcontroller applications.

### 7.2.3. System Calls

In order for programs to make use of the paging mechanism there must be a way for them to request more memory from the operating system. One such API is the `mmap` and `munmap` functions defined in the POSIX standard [20, 2, 3].

This standard provides an interface for mapping files and devices into the address space of a program. Most platforms also use this interface for allocating regular memory through the `MAP_ANONYMOUS` [21] option. The standard C library functions `malloc` and `free` are typically implemented using this API.

Using this API enables the programmer to use a known and well tested set of functions, once again making it simple to use. In this implementation the main focus is memory allocation, so options other than `MAP_ANONYMOUS` are not implemented for now.

```
1 void *mmap (void *addr, size_t len, int prot, int flags, int fildes, off_t off);
2 int munmap (void *addr, size_t len);
```

Listing 11: The memory mapping functions [2, 3]

Listing 11 shows the function prototypes that must be implemented for mapping regular, anonymous memory. Most of these may seem self explanatory however the flags and protections may be worth mentioning:

- `addr` is the virtual address where the memory will be mapped. Although sometimes useful for replacing page mappings or for other special use cases, leaving this zeroed lets the operating system select an address for the mapping by itself. For regular memory allocation, this should be zero so that a free portion of the address space is selected.

- `len` is the length, or size, of the memory mapping. Normally this must be a multiple of the page size and not zero, violating these conditions will cause the mapping to fail.
- `prot` specifies the access rights to the memory. Valid values are `PROT_READ`, `PROT_WRITE`, `PROT_EXEC` and `PROT_NONE` for read, write and execute permissions.
- `flags` specifies the kind of mapping to perform. Most implementations have many types of mappings with several options, however since this implementation focuses on allocation of regular memory only the `MAP_ANONYMOUS` and `MAP_PRIVATE` combination is used.
- The `fildes` and `off` arguments specifies the file and offset within that file to map when the operation applies to a file. These arguments are ignored for this implementation but are still there to allow expansion in the future and compatibility with the standard.

The standard requires the two function prototypes and any required flags to be located in `sys/mman.h`. Simple stubs are written that comply with these requirements, which stores the arguments and triggers a system call for the operating system to handle.

Calls to `mmap` and `munmap` end up in the kernel as `SvcMemmanHandleMmap` and `SvcMemmanHandleMunmap` respectively. Algorithm 6 lists the flow of the `SvcMemmanHandleMmap` function. A series of vailidity checks are first made to make certain that this is a supported call. Memory is then allocated with the memory pointer stored in `mem` and the calculated MPU register values in `mpu`. Using the protection attributes specified the register values are modified for correct operation using `SetRegionAttributes`. At last an entry is added to the pager system using the current PID, the calculated MPU registers, the data pointer that will be used by the program and the reference needed for freeing the resource later. The data pointer is then returned to the caller.

**Input :** addr ← The address of the memory
**Input :** len ← The size of the memory
**Input :** prot ← Access rights for the memory
**Input :** flags ← Type of mapping and options
**Input :** fildes, off ← Optional file and offset, ignored
**Input :** currentpid ← The process identifier of the currently running program
**Output:** MAP_FAILED with errno for failure or address on success

**if** *addr != 0* **then**
  **return** *ENOTSUP*;
**end**
**if** *len == 0* **then**
  **return** *EINVAL*;
**end**
**if** *IsInvalid(prot)* **then**
  **return** *EINVAL*;
**end**
**if** *prot == PROT_EXEC* **then**
  **return** *ENOTSUP*;
**end**
**if** *flags != (MAP_PRIVATE | MAP_ANONYMOUS)* **then**
  **return** *EINVAL*;
**end**

mem, mpu ← MemAllocMpu (len);

**if** *mem == 0* **then**
  **return** *ENOMEM*;
**end**

SetRegionAttributes (mpu, prot);

**if** *PagerAddEntry(currentpid, mpu.rbar, mpu.rasr, mpu.dataptr, mem) failed* **then**
  MemFree (mem);
  **return** *ENOMEM*;
**end**

**return** *mpu.dataptr*;

**Algorithm 6:** Memory management: SvcMemmanHandleMmap

The `PagerAddEntry` is responsible for finding a free entry in the page table and writing values corresponding to the structure in listing 11. Most of the page table entry values are simple set from the argument with the exception of `maxptr`. This value is used to check the upper bound of a region and depends on the values of the sub-region disable bits in the RASR. In order to not calculate this value for every page fault it is calculated by a helper function. Both the `PagerAddEntry` and the `PagerCalculateMaxAddress` are shown in listing 12.

```
int PagerAddEntry (unsigned int tid, unsigned int rbar, unsigned int rasr,
    unsigned int useptr, unsigned int freeptr)
{
    struct PageTable *table = &pagetable;

    do
    {
        for (int i = 0; i < PAGER_MAX_ENTRIES; i++)
        {
            if (table->entries[i].tid == -1)
            {
                table->entries[i].tid = tid;
                table->entries[i].useptr = useptr;
                table->entries[i].maxptr = PagerCalculateMaxAddress (
                    useptr, rbar, rasr);
                table->entries[i].mpu_rbar = rbar;
                table->entries[i].mpu_rasr = rasr;
                table->entries[i].freeptr = freeptr;
                return 0;
            }
        }
    } while ((table = table->next));

    return -ENOMEM;
}

unsigned int PagerCalculateMaxAddress (unsigned int base,
    unsigned int rbar, unsigned int rasr)
{
    unsigned int regionsizeexp = (rasr >> 1) & 0x0000001F;
    unsigned int regionsize = 1 << (regionsizeexp + 1);

    if (regionsize < 256)
        return base + regionsize;
    else
    {
        unsigned int subregionsize = regionsize / 8;

        for (int i = (1 << 8); i < (1 << 16); i = i << 1)
            if (!(rasr & i))
                base += subregionsize;

        return base;
    }
}
```

Listing 12: The PagerAddEntry and PagerCalculateMaxAddress functions

Note that the enabled subregions are always continuous and the base address is already set to the first enabled subregion. It is possible to simply loop over the bits and add the subregion size to get the correct end address of the region.

The process of unmapping memory follows a similar fashion, with pseudocode shown

in listing 7. A pager function called `PagerFreeEntry` is called to free any matching entry in the page table and free the memory. If no entry is found the function returns `EINVAL` indicating that the parameters supplied are incorrect. Otherwise the MPU registers and a scheduler yield is triggered to flush any old references to the memory.

**Input** : addr ← The address of the memory
**Input** : len ← The size of the memory, currently ignored
**Output:** Non-zero for failure or zero on success

**if** *PagerFreeEntry (currentpid, addr) failed* **then**
    **return** *EINVAL*;
**else**
    ClearCurrentMpuRegisters (currentpid);
    SchedYield ();
    **return** *0*;
**end**

**Algorithm 7:** Memory management: SvcMemmanHandleMunmap

### 7.2.4. Page Fault Handler

Since only two paged memory regions can be stored in the processor registers at any time, references to additional regions only stored in the table will trigger an access fault. When this happens the processor will go into the data abort mode of execution and start executing the abort handler. It is then up to this handler to find the correct page entry from the page table, load it into the hardware registers and restart execution of the failed instruction. The code that handles the event is typically called the page fault handler.

In this implementation the handler is split into three parts: The first is the abort handler called `DataAbort`. It is written in assembly to ensure that processor state is properly taken care of. It first stacks a set of registers that will be used and checks that the fault happened in a user program and is indeed an access fault. If it was not, the exception is assumed to be fatal. This may typically happen if the bad memory reference occurred in privileged code or there was a hardware error. It then calls the real `PagerWalk` function which does the actual search and sets the MPU registers once `PagerWalk` returns. The code for `DataAbort` is listed in listing 13, excluding the handling of fatal conditions. Note that this code is processor specific for the Cortex-R, the Cortex-M would be slightly different.

```
stmdb sp!, {r0-r6,lr}       @ Stack regs that will be overwritten
sub sp, sp, #12

mrs r1, spsr                @ Get the CPSR from the banked reg
and r1, #31
cmp r1, #16
bne PagerFailed             @ Processor mode is not user, always fatal

mrc p15, 0, r3, c5, c0, 0   @ Read the Data Fault Status Register
movw r1, #1039              @ 0x40F
and r3, r3, r1
cmp r3, #13
bne PagerFailed             @ Not a permission fault

ldr r4, =currentthc         @ r1 is currentthc pointer
ldr r1, [r4]
mov r2, sp                  @ r2 is scratch stack space
mrc p15, 0, r0, c6, c0, 0   @ Read the Data Fault Address Reg into r0

bl PagerWalk
cmp r0, #0
beq PagerFailed

ldr r3, =pagermpuregion
ldr r3, [r3]
ldm r0, {r4-r6}
mcr p15, 0, r3, c6, c2, 0   @ Set region
mcr p15, 0, r4, c6, c1, 0   @ Set address
mcr p15, 0, r6, c6, c1, 4   @ Set access rights
mcr p15, 0, r5, c6, c1, 2   @ Set size and enable

add sp, sp, #12
ldmia sp!, {r0-r6,lr}       @ Unstack
subs pc, lr, #8             @ Return
```

Listing 13: The `DataAbort` function

### 7.2.5. Table Walking Algorithm

The `PagerWalk` function is shown in listing 14. Its job is rather simple, it just loops through the blocks of page table entries looking for the entry with the right PID that has a matching address range. If one is found the job of carrying out any calculations are given to the third function, `PagerWalkHandle`. The `PagerWalk` function takes four

arguments: The address that just faulted, the control block of the program that triggered it, an array where MPU register values are returned and the Data Fault Status Register (DFSR). This register contains additional information about the fault such as if it was a read or write operation, later checked against the region permissions. If the handler function does not run into any errors, it returns the address of the output array which is then used to set the MPU registers, otherwise it returns zero.

```
1 unsigned int *PagerWalk (unsigned int address, struct ThreadControl *trc,
2     unsigned int *out, unsigned int dfsr)
3 {
4     struct PageTable *table = &pagetable;
5     unsigned int tid = trc->pctrl->pid;
6
7     do
8     {
9         for (int i = 0; i < PAGER_MAX_ENTRIES; i++)
10            if (table->entries[i].tid == tid &&
11                address >= table->entries[i].useptr &&
12                address < table->entries[i].maxptr)
13            {
14                return PagerWalkHandle (&table->entries[i], trc, out, dfsr);
15            }
16    } while ((table = table->next));
17
18    return 0;
19 }
```

Listing 14: The `PagerWalk` function

The `PagerWalkHandle` function is best explained using the pseudocode in algorithm 8. It is used to verify that the requested access type indeed matches what the page table entry specifies. If this is not done the page fault handler may end up in an infinite loop since the same handler is invoked for both non-existing entries and conflicting access types. Next the correct register values to set are calculated and stored in the `out` variable, which will be used by the first assembly function.

```
Input  : entry ← Page table entry to use
Input  : trc ← Thread control block
Input  : out ← Output array for new register values
Input  : dfsr ← Data Fault Address Register
Input  : pagermpuregion ← Index of the hardware region to set
Output: Pointer to the MPU registers or zero
if WasReadAccess(dfsr) then
    if ReadIsNotAllowed(entry) then
        return 0;
    end
else if WasWriteAccess(dfsr) then
    if WriteIsNotAllowed(entry) then
        return 0;
    end
end
CalculateAndStoreMpuRegisters (out, entry);
if pagermpuregion == PAGER_REGION_1 then
    SetRegionData (trc.mpuregs, PAGER_REGION_1, out);
    pagermpuregion == PAGER_REGION_2;
else
    SetRegionData (trc.mpuregs, PAGER_REGION_2, out);
    pagermpuregion == PAGER_REGION_1;
return out;
```

**Algorithm 8:** Memory management: PagerWalkHandle

### 7.2.6. Replacement Policy

Perhaps more interesting is the last part of the `PagerWalkHandle` function in algorithm 8, which is also responsible for the replacement policy. As previously discussed only two hardware entries are used due to simplicity. With only two entries the task of tracking the previously used entry becomes rather simple, all that needs to be done is to store the index of the one that was last set and alter between them. This index is stored in the global variable `pagermpuregion` and then used by the assembly code when setting the real registers. Note that since the variable is global, multiple programs running at the same time may interfere when using paging. The issue is again left for later investigation as it is not the main focus here.

### 7.2.7. Miscellaneous Page Table Functions

The paging system requires additional functionality to be really useful and stable. These components ensure that memory is properly managed by preventing leaks when programs are terminated.

- `PagerInitialize` is called once at boot time to initialize the page table with cleared entries.
- `PagerCleanupByPID` deletes all entries in the page table associated with the given PID. It is used when a program is forcefully terminated by an access fault or the user by signals or the common `Ctrl+C` keyboard command. Running this function prevents the system from leaking memory references and inevitably locking up at a later time.

# 8. Testing and Verification

In this chapter the solutions presented in chapter 5, 6, and 7 are tested and verified first through functionality testing and then through a practical example. The results of these tests makes it possible to evaluate the solution(s) against the research questions and see if they have been fulfilled.

## 8.1. Memory Allocation

The testing and verification of the memory allocation algorithms are detailed in this section. A brief discussion of its performance characteristics is first performed before a set of functional tests. Note that further testing is also performed in the next major sections as they depend on correct operation of the allocator.

### 8.1.1. Performance Considerations

Due to the central importance of the memory allocator an analysis of execution time is appropriate. Use of the allocator may be split into two situations: the time when an application is started and the time when an application is running. The latter situation is often much more sensitive to performance issues than the former. In any case, developers often desire fast operation that executes in near constant time.

The code listed in appendix B.5 is used as a basis for analysis as well as the pseudocode listings in chapter 5. From code analysis, the following is determined:

- Determining the amount of free memory: $O(1)$ time. The implementation maintains an integer value describing the absolute amount of free memory at any given time.
- Determining the lowest amount of free memory (historical): $O(1)$ time. The value is maintained in the same manner as with the current amount of free memory.
- Searching for a free memory block: $O(n)$ time. The internal list of blocks has to be traversed in order to find a free block which is a linear operation. At the cost of using more memory this may be improved by storing internal data in linked lists instead of linear arrays.
- Searching for a free table entry: $O(n)$ time. Once a memory block has been found its table entry must be split into the to be occupied portion and the remaining portion. This requires a new table entry, which must be searched for. Improvements to this linear time requirement may also be made using linked lists at the cost of memory consumption.

- Finding the allocation entry for a given memory block by address: $O(n)$ time. Again the internal list of memory blocks must be traversed. Here, potential improvements may be done using tree structures, which would increase the memory consumption even further.

- Allocating memory: $O(2n)$ time. The allocator will first have to loop for a large enough free block, and then for a free metadata table entry. Improvement depend on those two operations.

- Freeing memory: $O(2n)$ time. The deallocator will first have to find the entry corresponding to the address and then coalesce any references to it. As with allocation, improvement depend on data structure.

It is clear that the allocator and deallocator are not the most suitable for speed or determinism. It should, therefore, be used for large allocations during application setup. Where performance and/or determinism is required a small local heap with a suitable allocation algorithm is the preferred method. This smaller heap may then be allocated by the slower during setup. Typical implementations of the standard `malloc` and `free` library functions often keep pieces of memory in bucket lists that allow for very fast allocations. In any case, execution time of the allocator is linear which is good enough for most applications.

Since the allocation algorithms are almost exclusively used during program loading, the impact of the linear search times are not very significant during real world usage. This is mainly for two reasons: The loading will normally happen when the end application is started, when the system is powered up and a few extra milliseconds does not matter, and because loading times are dominated by disk access times.

It is reasonable to conclude that the research questions 2.1 and 2.2 are well satisfied in the context of the allocation algorithm itself. The question still remains whether the protection hardware itself or any other surrounding software primitives make a significant impact. This is discussed in the following sections.

### 8.1.2. Functionality Testing

As a test method for the memory system, functionality testing is selected. With this type of testing each function of the software is tested for correct operation against the requirements of the software, in this case the requirements of the MPU hardware. Software functions are tested as black boxes, that is, the testing does not concern itself with the actual source code but the input and output data instead. In order to get better results however some knowledge of the code is used in order to increase the chance of finding a fault.

The goals of the functionality testing is to check that

- The calculated register values are valid and correct for the hardware, which will indicate that the solution is at all usable.

- That none of the allocations give unintended access to others by overlapping memory regions.
- That the solution is indeed able to satisfy research questions 1.1 to 1.3.

### 8.1.3. Test Subjects

The functions targeted by the testing are those that are part of the memory management system, used during allocation and de-allocation:

- `MemAlloc` - Plain memory allocation
- `MemAllocMpu` - MPU enabled memory allocation
- `MemFree` - Memory de-allocation for both of the above

As a consequence of testing these functions, some sub-routines are also tested as the above depend on them:

- `MemMerge` - Merging of two free blocks
- `MemAllocMpuCalcRASR` - Calculation of the RASR
- `MemAllocMpuFindMinimal` - Finding the closest power of two
- `MemAllocEntry` - Allocating an entry
- `MemSplit` - Splitting an entry

In order to get good results, as much of the code functionality as possible must be tested. However since the test method is functionality testing and not unit testing, only general exercise of the above functions should be performed.

### 8.1.4. Methodology

Since the behaviour of memory allocations depend on previous state, all of the functions must be tested together and in different ways to expose problems that depend on that state. With memory allocation, state mostly refers to current and previous allocations but may also contain memory leaks and fragmentation. In order to avoid such issues the testing is performed right after the memory system is initialized at startup time. For this test a total memory size of at least 16 kilobytes is assumed.

The following test procedure is proposed:

1. `MemAlloc` of 1000 bytes
2. `MemAlloc` of 1000 bytes
3. `MemAllocMpu` of 7000 bytes

4. `MemFree` of step 2
5. `MemAllocMpu` of 512 bytes
6. `MemAlloc` of 200 bytes
7. `MemFree` of step 1
8. `MemFree` of step 3
9. `MemFree` of step 5
10. `MemFree` of step 6

This procedure will exercise all of the three target functions in various orders while also exercising the sub-routines listed earlier. Memory blocks are expected to be allocated, split, merged and freed while also testing for MPU rules.

### 8.1.5. Expected Results

In order to calculate the expected results, it is important to remember that the actual starting address of the memory pool may not be neatly aligned. As a consequence the `MemAllocMpu` function may add some padding to its allocations.

The expected outcome of the test-run is as follows:

1. An address for the very first 1000 bytes is returned, and the internal tables now have two entries: one for the allocation and one for the free memory.
2. The very next 1000 bytes is returned and the internal table is split once more.
3. Since the previous allocations does not align to a power of two address, an allocation that skips a portion of memory is expected in this case due to the MPU specification. A region size of 8192 is expected as it is the closest power of two. Since the allocation allows for one subregion to be disabled it is expected to be a disabled subregion at either the start or the end of the allocation. Since the memory must be isolated an actual allocation of at least $8192 - 1024 = 7168$ bytes must be made.
4. The block previously allocated in step 2 is marked as free, and possibly merged with any padding done in step 3.
5. An allocation in the space previously held by step 2 is expected, with no subregions disabled as the requested size fits a power of two.
6. The return address is expected to be right after the allocation in the previous step.
7. The block is marked as free.
8. The block is marked as free and merged with the rest of the memory pool.

9. The block is marked as free and merged with the rest of the memory pool.
10. The block is marked as free and merged with the rest of the memory pool.

At the end of the procedure the internal tables must have the same state as they did before the start of the procedure. Failure to do so would indicate a cleanup or fragmentation problem. Furthermore no overlapping allocations must occur at any point in time during the test.

### 8.1.6. Test Execution

For best results the test is executed by the target hardware right after the memory system is initialized. This makes it more likely to detect problems that may be hardware dependent, and executes the tests before any memory is put to use. The test code is listed in appendix B.1. After executing the test, the following result output can be observed:

```
Added 174780b addr 0x08005544
Step 1: Address 0x08005544
Step 2: Address 0x0800592C
Step 3: Address 0x08005D14
   dataptr 0x08006000
   rbar 0x08006000
   rasr 0x00008018
Step 4: Done
Step 5: Address 0x0800592C
   dataptr 0x08005A00
   rbar 0x08005A00
   rasr 0x00000010
Step 6: Address 0x08005C00
Step 7: Done
Step 8: Done
Step 9: Done
Step 10: Done
```

### 8.1.7. Discussion

As can be seen first in the previous listing, 174780 bytes of memory was added to the pool starting at the address `08005544h`. As expected the first allocation is assigned to the very first bytes of the pool. The next 1000 bytes happen right after at the address $08005544_{16} + 1000_{10} = 0800592C_{16}$.

Next up is the first MPU enabled allocation. At this point the free memory space starts at $0800592C_{16} + 1000_{10} = 08005D14_{16}$. There are four variables returned by the allocator:

- The address of the entire block that has been allocated, and that is used for freeing the memory later.
- The starting address of the block that may be protected by the MPU. This is the lowest address that a program with its access restricted may use without triggering a fault.
- The value to be programmed into the RBAR, following the definition in figure 5.4.
- The value to be programmed into the RASR. It follows the definition in figure 5.5 setting the specified sub-region disable bits and size bits. With the value of $8018_{16}$, one of the sub-regions has been disabled as expected to save that extra memory. The size is correctly set to 8192 bytes which leaves an accessible 7168 bytes accessible to the program. Note that the reserved bits shown in the figure contains access rights and cache information which must be set accordingly later.

The allocator has padded the memory block accordingly to get an aligned starting address for the MPU. It has then assigned 7 sub-regions for an actual allocation of 7168 bytes which is the smallest size that will fit the requested 7000.

In the next two steps the memory starting at $0800592C_{16}$ is freed before a new MPU aware allocation is made. This allocation is expected to lie in the memory just freed, before the previous MPU allocation. Indeed, an address identical to what was previously freed has been given. Since the allocation aligns to an entire MPU region no sub-region bits has been disabled however some padding has been added at the beginning to satisfy the alignment.

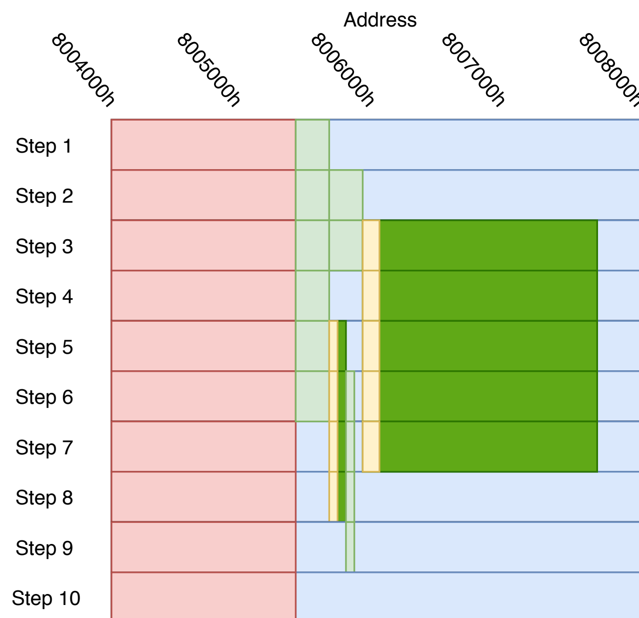


Figure 8.1.: Memory Allocation Test Progress

At last an allocation is made which should be placed in-between the two MPU enabled allocation. This is indeed the case, and the memories are then freed. A visual presentation of the allocation progress can be seen in figure 8.1.

The figure only highlights the first interesting part of the memory map and is not to scale. Read area are memory outside of the allocation pool. Light green represents regular memory while dark green represents MPU enabled memory. Yellow area is memory used for address alignment.

### 8.1.8. Conclusion

At this point it can be concluded that the implementation is functional and satisfies the hardware requirements. Any further testing would require more in-depth techniques such as unit testing and integration testing. Further functionality testing is done in the remaining parts of this chapter where the allocation system is actually used.

As the algorithms generated the expected protection values, there is no reason to expect that research questions 1.1 to 1.3 will not be fulfilled. Correct usage of the algorithms does however depend on the loader which is evaluated in the next section. Therefore, the full conclusion is drawn in the next sections.

## 8.2. Program Loading

This section briefly goes through functional testing of the program loading routines. The subject of the testing is the shell, a program that takes textual user input and interprets their commands, launching the programs specified. The loading subsystem is responsible for making correct usage of the allocation subsystem, making correct operation essential for achieving the goals outlined in any of the research questions.

### 8.2.1. Performance Considerations

The `posix_spawn` implementation is one of the most complex calls in the kernel, pulling strings in major departments: scheduling, file systems, allocation and deferred IO among others.

Due to the amount of disk accesses that has to be performed especially during relocation, considerable time may be consumed. This is in the nature of disk usage since they are often indirectly accessed block devices and may even be mechanically limited, which is the case for hard drives. Although only a fraction of a second may be spent this is sufficient to cause problems for concurrent real-time applications that depends on interrupt deadlines. This is solved by deferring the call to the `posix_spawn` implementation: the caller is simply suspended from execution and the implementation then runs in a proper thread that may be scheduled and de-scheduled along with other programs. Other programs are then not blocked from execution and may operate while the new program is being loaded. Once the load is complete the worker thread stops and the caller is allowed to resume.

### 8.2.2. Test Subjects

The functionality to be tested is the implementation of the standard POSIX calls `posix_spawn` and its supporting functions: `waitpid`, `kill`, `exit`, `getpid`, `getpgrp` and `setpgrp`. These are all fundamental for process loading and management. Their responsibilities in a system are as follows:

- `posix_spawn`: Load a specified program image into memory and start its execution. In addition it allows the caller to specify the program arguments and environment, actions to perform on open file handles, and attributes to set for the program.
- `waitpid`: Wait for a program to change state. This call is capable of both blocking and non-blocking execution, that is if it waits for an event to happen before returning or not. State changes are typically caused by programs exiting, crashing or being suspended.
- `kill`: Sends a signal to a process or process group by their numerical identifier. This call is typically used for teminating a process, hence the name, but also for suspending, resuming and general communication.
- `exit`: Used by a program that terminates by itself, either by returning from the main function or calling `exit` explicitly.
- `getpid`: Used by a program to get its own process identifier.
- `getpgrp` and `setpgrp`: Get and set the process group. Process groups are typically used to create a logical group of processes that are part of one job that may then be used to suspend, resume and signal the entire job in one operation. Process groups are also used to identify which programs that are considered to be "in front", that is, allowed focus of the user.

### 8.2.3. Methodology

In order to test process loading and inter-process interaction the shell is used to perform the tests. The shell is written to support most of the typical job-control use cases including single-process jobs with Input-Output (IO) redirection and background processing. Since the functionality is fairly standard it is beneficial to select a test procedure that will also function on a standard PC to establish a baseline:

1. Test the ability to start a simple program such as a basic "hello world" program
2. Test IO redirection by piping the "hello world" program into a file and then piping that file back into a processing program such as grep.
3. Test background processing by starting a program in the background, then starting a program in the foreground, let the foreground program finish and then resume the first program in the foreground before terminating.

4. Test terminating a program explicitly using the `kill` function.

A shell is a very complex program which is mostly irrelevant to discuss here. However, a listing of the function that performs the final calls into the `posix_spawn` API is listed in appendix B.2. The full version of the shell is supplied in the archive given with the thesis, and can be found in the `bin/sh` directory. In addition to just program setup the shell also handles

- Terminal setup
- Displaying and processing the command prompt
- Command tokenization
- Auto-completion of file paths
- Environment variables
- Command history
- Job control

All of these functions except for command history and auto-completion is exercised in the outlined test procedure.

### 8.2.4. Expected Results

The expected result is that the tests run on the desktop machine and those run on the microcontroller show identical behaviour. That is, they both show the same capabilities regarding starting a program, redirecting its input and output channels, suspending programs and resuming programs.

### 8.2.5. Test Execution

The results shown in figure 8.2 are obtained by running the above test procedure on a GNU/Linux desktop machine in a terminal. The shell program previously mentioned, written for microcontroller use, is compiled for the desktop machine and used in the test. Strictly speaking the native `bash` shell could be used, however by using the same shell on both systems the results will be easier to compare.

```
ejs /home/erlendjs/documents/git/orbit/programs/helloworld/build
$ ./helloworld
Hello, World!
ejs /home/erlendjs/documents/git/orbit/programs/helloworld/build
$ ./helloworld > testfile
ejs /home/erlendjs/documents/git/orbit/programs/helloworld/build
$ grep Hello < testfile
Hello, World!
ejs /home/erlendjs/documents/git/orbit/programs/helloworld/build
$ sleep 10 &
ejs /home/erlendjs/documents/git/orbit/programs/helloworld/build
$ ./helloworld
Hello, World!
ejs /home/erlendjs/documents/git/orbit/programs/helloworld/build
$ fg
[1] (19122) Done sleep
ejs /home/erlendjs/documents/git/orbit/programs/helloworld/build
$ sleep 20 &
ejs /home/erlendjs/documents/git/orbit/programs/helloworld/build
$ jobs
[1] (19127) Running sleep
ejs /home/erlendjs/documents/git/orbit/programs/helloworld/build
$ kill -9 19127
[1] (19127) Done sleep
ejs /home/erlendjs/documents/git/orbit/programs/helloworld/build
$ jobs
ejs /home/erlendjs/documents/git/orbit/programs/helloworld/build
$ 
```

Figure 8.2.: Testing the process loading on the desktop

The following figure 8.3 is the result of executing the procedure on the target microcontroller. Keep in mind that the syntax is slightly different due to folder structure and shell behaviour.

```
ejs /test
$ helloworld
Hello, World!
ejs /test
$ helloworld > testfile
ejs /test
$ grep Hello < testfile
Hello, World!
ejs /test
$ sleep 10 &
ejs /test
$ helloworld
Hello, World!
ejs /test
$ fg
[1] (28) Done sleep
ejs /test
$ sleep 20 &
ejs /test
$ jobs
[1] (30) Running sleep
ejs /test
$ kill -9 30
[1] (30) Done sleep
ejs /test
$ jobs
ejs /test
$
```

Figure 8.3.: Testing the process loading on the microcontroller

### 8.2.6. Discussion

As shown in the previous section with figures 8.2 and 8.3, the results are the same for both the desktop baseline and the test using the microcontroller system. There are however some differences. The first difference is that the syntax is slightly different between the two due to using different folder structures. In both results the same shell is used, originally written for low resource microcontroller use but cross-compiled for both platforms and run without modification in both cases. In the baseline example which is run on a GNU/Linux system the grep and sleep programs are from the standard GNU core utilities while the microcontroller versions are minimal implementations. The hello world program is cross-compiled for both systems just like the shell.

### 8.2.7. Conclusion

Running the same software on both platforms without modification is only possible due to standard interfaces, and the test proves the advantage of relying on those standards. As long as compatibility is maintained across as many function calls as possible, developer effort will remain low.

Getting identical results from both the baseline and microcontroller tests reinforces the idea that the implementation is fundamentally correct and in conformance with the POSIX standard. It does however not guarantee that the implementation is bug free and complete. Testing for such issues would require a more detailed approach using unit

and integration testing among others.

Testing of crash handling is purposefully held out of this test run as it is tested in section 8.3, which relates to research questions 1.1 to 1.3. As previously discussed the loading of a program has performance issues due to the disk accessing done and processing work needed. The loading is done in a separate thread allowing other programs to function normally, however if the program requesting the loading itself is subject to real time performance constraints there is likely to be issues. This relates to research question 2.1. In other words, this research question is partially affected by the solution. The impact of the time needed to perform the loading can however be mitigated:

- Software with strict real-time demands can be run in a separate process from the start or in the kernel itself. This does however require that the limitations are taken into account when designing the application.

- A faster disk can be used so that the loading time is also reduced.

In any case the problem remains possible to solve as long as the time consumption is taken into account. The same problem applies to other platforms as well, such as Linux and Windows since starting a program also involves disk accesses and processing work on these platforms.

## 8.3. Paging and Memory Protection

This section adds to the previous by testing the pager described in chapter 7. The pager relies on the memory allocation described in chapter 5 and tested in section 8.1. The crash and error handling relies on the process loading mechanisms described in chapter 6 and tested in section 8.2. The test of the pager finally tests the systems ability to recover from program crashes and remaining stable, allowing research questions 1.1 to 1.4 to be fully evaluated. That is, the systems ability to detect and handle memory violations while remaining operational, and to do so without added developer effort.

### 8.3.1. Performance Considerations

As previously noted the pager only uses two hardware entries, which is the least amount that allows copying between two memories without frequent page faults.

Since protection regions in the MPU hardware is not limited to a fixed size and most microcontroller software makes scarce use of dynamic allocation, these two entries may often satisfy the needs of a program. When more allocations are made the performance is expected to drop due to page faults, subject to the access locality of the program. That is, the pattern of memory accesses performed by the program.

The program `crashtest` in the `usr` directory may be used to test the behaviour of the pager. When accessing one or two allocated memories, page faults only occur for the first access. Any further accesses run at the highest performance.

With the current implementation not written for raw performance, the perceived speed of the program may drop significantly when more than two allocations are made. There are multiple ways to improve this:

- Improve the structure of the page table. Data may be stored in separate tables per process so that the PID value is removed from the entry. Furthermore the table may be converted into a tree structure, which may require some effort due to the many possible region sizes.
- Write a larger portion of the page fault handler in optimized assembly code. This has drawbacks however, such as producing less portable code and requiring more effort both during initial development and maintenance.
- Using larger and fewer allocations in the user program, which will in turn reduce the pressure on the abort handler.
- Use more hardware entries, subject to the number of entries available. Since the number of available entries depends on the silicon manufacturer, the actual number of regions available for paging may be as low as four regions in total.

Recall that the four regions required by the program text, read-only data, stack and read-write data is not swapped or modified by the paging mechanism. This guarantees that execution of programs that do not make use of the pager will run without page faults and by that have optimal performance.

In general, very large and complicated software is likely to not run well on a microcontroller and run best on a proper application processor which implements a full virtualized memory solution not subject to the limitations discussed here.

As for the performance-related research questions, it is clear that they will be fulfilled as long as the hardware used is properly selected for the job.

### 8.3.2. Test Subjects

The testing outlined in the following sections target the `mmap` and `munmap` functions that handle allocation requests, the fault handler for illegal memory accesses, and the entire systems' ability to handle crashed programs.

### 8.3.3. Methodology

A special test program is written to exercise the system components. It is compiled as a regular program and started with the shell used in the previous test, building upon that test by relying on the same systems.

The purpose of these tests is to verify that the solution(s) successfully accomplish the goals and challenges outlined in research questions 1.1 to 1.3. That is, the ability to detect and handle memory access violations. A successful test indicates that the allocation scheme, program loader and paging mechanism is not giving access to data that it should not. They also show that they do give access to areas that they should,

since the tests would otherwise not run at all. There are five tests written to exercise different parts of the memory protection and crash systems:

1. A basic null pointer exception test. The null pointer is a memory pointer with a value of zero, indicating that it is invalid. Although many systems have actual memory or hardware mapped to the first area of the memory map this is usually reserved and should not be accessible to user programs. Dereferencing a null pointer is typically associated with software bugs and should cause an exception that is either handled or used to terminate the offending program.

   The first test simply attempts to read the 32-bit word at address zero. If the test is successful a crash should be reported and the program should not be allowed to continue.

   The C code function for the null pointer test is listed in appendix B.3, lines 40 to 54.

2. A universal memory access test. This test is similar to the null pointer test except that the address to read from is taken as an input argument. It is useful for verifying that hardware registers for peripherals and IO is properly protected. For this, two addresses are tested: the start of the physical RAM which is used for kernel working memory at address $08000000_{16}$, and the system clock speed control register at $FFFFFF70_{16}$.

   The C code function for the universal pointer test is listed in appendix B.3, lines 56 to 89.

3. Single `mmap` and `munmap` with access tests. This is the first test that interacts with the paging mechanism which is only used for `mmap`-ed memory. A memory block of 1024 bytes is requested from the system with read and write access rights. The memory is then written and read to, which verifies that the allocation has indeed been assigned to the MPU hardware, so that the program may access it. Since the hardware registers are set on the first access and not during allocation this will trigger a page fault, and so the test verifies that the page fault handler is functional and able to locate and set the correct page.

   The C code function for the single allocation test is listed in appendix B.3, lines 91 to 118.

4. Double `mmap` and `munmap` with access tests. As previously discussed, two hardware entries are used to implement the paging mechanisms. This allows a maximum of two memory regions to be “open” at the same time with additional regions swapped in by the pager when necessary. By testing with two regions the ability of the pager to operate with two entries at the same time is verified. When executing the test it is important to measure the number of page faults, since excessive faults indicates that either only one entry is used or that the program is experiencing interference with other programs.

The double allocation test operates in the same manner as the previous single allocation test with the addition of a memory copy and comparison operation.

The C code function for the double allocation test is listed in appendix B.3, lines 120 to 156.

### 8.3.4. Expected Results and Test Execution

The code listed in appendix B.3 is compiled and executed on the target microcontroller. Note that the crash messages are greatly simplified, only the most important information is shown. The results for the tests are listed below and discussed in the next section.

#### Null Pointer Test

The null pointer exception test is expected to trigger a crash of the test program. The shell that started the program should detect this and return to the command prompt, allowing new commands to be issued. The contents of the address zero should not be read by the program.

```
$ crashtest nullptr
Testing access violation
Crash detected!
Userspace
Data abort
pc  0x0800E754
dfsr  0x0000000D
dfar  0x00000000
$
```

As can be seen from the result, the access violation was successfully detected. The supplied register information indicate that the loaded program tried to access location zero.

#### Specified Pointer Test

The universal memory access test is expected to crash when specifying any address that does not belong to the test program itself, regardless of what type of memory the address is assigned to in hardware. As with the null pointer test the shell should return to the command prompt and the system must remain stable.

```
$ crashtest ptr 08000000
Testing pointers
Crash detected!
Userspace
Data abort
```

```
pc  0x0800E754
dfsr  0x0000000D
dfar  0x08000000

$ crashtest ptr FFFFFF70
Testing pointers
Crash detected!
Userspace
Data abort
pc  0x0800E754
dfsr  0x0000000D
dfar  0xFFFFFF70
$
```

As with the previous test, the results show that the illegal access was detected and handled. Notice that in both cases the value of the Data Fault Address Register (DFAR) is the same as the one supplied as an argument to the program.

### Results from the Single Allocation Test

The single allocation test is expected to perform a successful allocation, write and then verify the contents of the given memory, exercising the pager in this process. At last the memory is to be unmapped. A page fault is expected to occur with the first access to the allocated memory.

```
$ crashtest singlemmap
Allocating space: 1024 bytes
Interacting with memory...
PagerWalk 0x08005800 pid 3
PagerWalk 3 08005800 00000013 00001306
Unmapping
$
```

The results show that the pager is used, with the `PagerWalk` messages being printed by the paging system. A single page was allocated and loaded into the hardware registers as indicated by the page walk occurring only once. Note that in contrast to the previous results the program did not crash, meaning that the program was allowed access to the new memory and that none of the base allocations were disturbed.

### Results from the Double Allocation Test

The double allocation test should operate in the same way except that it should trigger two page faults, and only two. Once both of the allocations has been assigned to

hardware protection registers there should be no further page faults and execution speed must not be hindered.

```
$ crashtest doublemmap
Allocating space: 1024 bytes x2
Interacting with memory...
PagerWalk 0x08005800 pid 4
PagerWalk 4 08005800 00000013 00001306
PagerWalk 0x08005C00 pid 4
PagerWalk 4 08005C00 00000013 00001306
Unmapping
$
```

The results indicate the same as the previous single allocation test, except that we now see two page walks since two blocks of memory was allocated.

### 8.3.5. Discussion

As can be seen by the results all of the test for the null pointer and chosen address tests result in a crash as expected. The program is denied access to those addresses and is forced to terminate.

The mini-dump shown by the kernel informs us about where in the code the fault happened and the address that the program tried to access: The "pc" is the program counter value at the time of the crash and will point to the faulting instruction.

The "dfsr" is the data fault status register which gives additional information about the type of fault, in these cases they are all read access faults. The "dfar" is the data fault address register which is the same as our supplied address in all cases indicating that it did indeed attempt to access the specified memory location. In all cases the crash is detected by the shell which then returns to displaying the command prompt, allowing further commands to be specified.

At last there is the `mmap` tests. They both succeed without error. The first test triggers only one page fault while the second triggers two as expected. These are shown by the kernel printing the `PagerWalk` messages, which are manually enabled for the purpose of this test and not normally shown. For all the message pairs the first indicates the requested address and the process identifier of the program that attempted the access. This identifier is considered when walking the table to ensure that a process is only granted access to its own pages.The second message in each pair is shown whenever a page is found and includes the process identifier and the register values that will be written to the MPU hardware before the program is allowed to continue. The program is always resumed with the instruction that previously failed so that the page fault is completely transparent.

### 8.3.6. Conclusion

In all cases the results are as expected. The tests all show that programs are only allowed to access memory that they have been granted access to and that they may request more memory from the operating system when needed. This means that research question 1.1 is fully covered, since a write to the memory of a different program will not trigger a fault in that program but in the one that caused the fault in the first place.

As for research question 1.2, regarding security, the results show that even if the system is breached the attacker will have limited access. If even only a single access violation is made the attacked process will be terminated. Although this could be used as a denial of service attack, it prevents the program from leaking any information.

Research question 1.3 can be considered to be partially fulfilled: hardware failures that affect the internal operations of a program will be detected. However, hardware errors that have an effect on the operating system itself may go undetected or cause system instability. In order to cover the system itself, special error detecting hardware must be used.

In addition to just functionality, the memory mapping tests show that the paging scheme is not triggering any more page walks than necessary. This is important for research question 2.2.

## 8.4. Small System Use Case Example

In this section a small system use case is presented that illustrates the usefulness of strict memory protection running on a microcontroller. The application solution is split into three parts: The first is a main program that serves as the manager and overlooks the others, the second is a program that performs various calculations outputting result events, and a third program performs logging. The purpose of this test is to verify that the solution does indeed function in a real world use-case and strengthen the basis of discussion surrounding the research questions.

### 8.4.1. Methodology

The idea is that with the important calculation processing and logging code split into two parts, a fault in one will not cause trouble for the other. This reinforces the confidence in research question 1.1 to 1.2 being fulfilled. The manager is used to start the two programs and handle any crashes and errors that may occur. In this case the manager may also send messages to the other programs to demonstrate errors, which will trigger the error handling mechanisms. Named pipes are used for communications: one between the processing program and the logger, and two for the manager to communicate with them. Crashes are detected using the standard `waitpid` call and programs are started using the `posix_spawn` function.

The processing program perform the important work of the application. Since this is a showcase application, it only performs dummy operations as a placeholder for a real program. It outputs a number sequence at an interval that are to be logged by the

logging program, while checking for messages from the manager. If the manager signals that it wants to simulate a crash the processing program will terminate when receiving the message.

At last the logging program listens for results from the processing program and formats them into time stamped messages. In a similar fashion to the processing part, it listens for messages from the manager in order to trigger an induced fault.

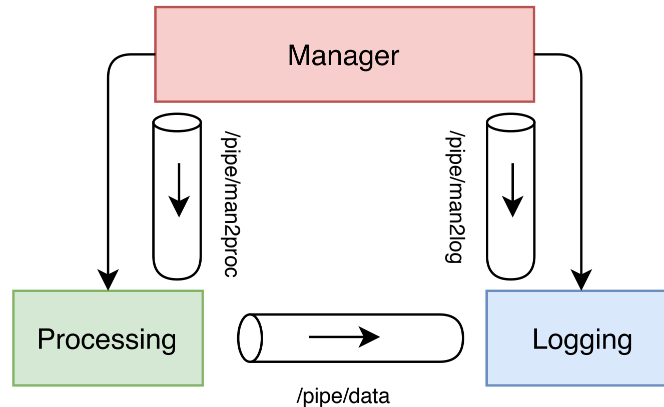


Figure 8.4.: Example Application Setup

Figure 8.4 illustrates how the programs fit together. The arrows shows the parent-child relationship between the processes, and the communication pipes are shown with direction and name.

### 8.4.2. Test Execution

The three programs are compiled into a file that are transferred to the microcontroller target filesystem. Using the shell, the manager program is started, which in turn starts the processing and logging programs after creating the pipes. After setting everything up it listens for keyboard input: the character `p` for testing a fault in the processing program and the character `l` for testing a fault in the logger. When a program exits, it is restarted after a few seconds. The code for the manager is listed in appendix B.4.

The processing program outputs a number sequence in a loop with a one second delay. These numbers are written to the pipe at standard output connecting the processing and logging programs. Meanwhile the pipe connecting the manager to the processing program is checked for data, where data indicates that a fault is to be simulated. The fault is simulated by executing an illegal memory access causing it to be terminated. This is similar to the testing done previously, exercising the same systems. The code for the processing program is listed in B.4.

At last, the logging program listens for data coming from the processing code. For each line it receives a timestamp is added before outputting to standard output. As with the processing program, a pipe is checked for commands that are used for simulating faults. To show that the logger is still operational when the processing program is not

present, a message is printed when output from it is missing for some time. The code for the logging program is listed in B.4.

Using the fault injection mechanisms, a fault is first simulated in the processing code and then the logging code.

### 8.4.3. Results

#### Load Addresses

The memory blocks that have been allocated to the programs can be printed by the kernel when enabled with the `sys showrelocs 1` command. Using this command it can be shown that the memories allocated to the programs are separate and non-overlapping:

```
text   at 0800E000 entry 0800EC1D
rodata at 0800F800
data   at 08010000
bss    at 0801042C
text   at 08012000 entry 08012C1D
rodata at 08005A00
data   at 08014000
bss    at 0801442C
text   at 08015000 entry 08015C1D
rodata at 08016800
data   at 08017000
bss    at 0801742C
```

The first block is for the manager process, the second for the processing process and the third block is for the logger process.

#### Processing Fault

A fault in the processing code is initiated by sending the p character to the manager, which in turn sends it to the processing code using the pipe. An access fault is then triggered causing the processing program to terminate abruptly. With the manager waiting before restarting it there will be a period where the logger does not receive data and outputs a message. The processing program is then restarted and normal operation resumes.

Successful execution of this test shows that all of the memory allocation, logging and paging mechanisms function properly, including the supporting infrastructure such as basic process management.

```
1 $ masters manager
2 Starting the logger
3 Logger got msg at 0: Processing starting, state: 0
```

```
4 Processing state: 1
5 Logger got msg at 2: Processing state: 2
6 Logger got msg at 4: Processing state: 3
7 Logger got msg at 6: Processing state: 4
8 Logger got msg at 8: Processing state: 5
9 Logger got msg at 10: Processing state: 6
10 Sending message to processing
11 Logger got msg at 12: Processing state: 7
12 Crash detected!
13 The processing program has terminated
14 Restarting processing in 6...
15 Userspace
16 Data abort
17 ... (Register dump not shown)
18 Logger got msg at 14: Processing got signal to crash
19 Restarting processing in 5...
20 Restarting processing in 4...
21 Logger warning: no work after 4 iterations
22 Logger warning: no work after 5 iterations
23 Restarting processing in 3...
24 Logger warning: no work after 6 iterations
25 Logger warning: no work after 7 iterations
26 Restarting processing in 2...
27 Logger warning: no work after 8 iterations
28 Logger warning: no work after 9 iterations
29 Restarting processing in 1...
30 Logger warning: no work after 10 iterations
31 Logger warning: no work after 11 iterations
32 Logger warning: no work after 12 iterations
33 Logger got msg at 27: Processing starting, state: 0
34 Processing state: 1
35 Logger got msg at 29: Processing state: 2
36 Logger got msg at 31: Processing state: 3
37 Logger got msg at 33: Processing state: 4
38 Logger got msg at 35: Processing state: 5
39 Logger got msg at 37: Processing state: 6
```

The results are color coded. Regular black text are used for messages from the manager program. Blue text is from the logger with cyan being the text the logger received from the processing program. At last the red text is the crash message from the kernel, simplified in this listing.

Note that the messages are not always in order. This is due to each program running in its own threads including the kernel thread printing the minidump. Additionally, the crash dump is truncated into a single line for simplicity.

### Logging Fault

A fault in the logging code is initiated by the character `l` and sent to the logger. Once the logger crashes, there will be no program to read the results from the processing code and data will build up in the pipe. When restarted, the logger will once again start reading and then print results rapidly to catch up.

The results from this test follow the same color coding and semantics as the processing fault test.

```
$ masters manager
Starting the logger
Logger got msg at 0: Processing starting, state: 0
Processing state: 1
Logger got msg at 2: Processing state: 2
Logger got msg at 4: Processing state: 3
Logger got msg at 6: Processing state: 4
Logger got msg at 8: Processing state: 5
Logger got msg at 10: Processing state: 6
Sending message to logger
Logger got signal to crash
Crash detected!
Userspace
Data abort
The logger program has terminated
Restarting logger in 6...
... (Register dump not shown)
Restarting logger in 5...
Restarting logger in 4...
Restarting logger in 3...
Restarting logger in 2...
Restarting logger in 1...
Starting the logger
Logger got msg at 0: Processing state: 7
Processing state: 8
Processing state: 9
ProLogger got msg at 1: cessing state: 10
Processing state: 11
Processing state: 12
ProLogger got msg at 2: cessing state: 13
Logger got msg at 3: Processing state: 14
Logger got msg at 5: Processing state: 15
Logger got msg at 7: Processing state: 16
Logger got msg at 9: Processing state: 17
```

### 8.4.4. Discussion and Conclusion

As can be seen by the results, the system is able to continue operating even after multiple program crashes. This is made possible by the careful cooperation of the memory protection, process loading and paging systems. Once again it reinforces that the implementing solutions are capable of handling a real user scenario, reinforcing research questions 1.1 to 1.3.

In addition to the first three questions, this test also provides a means to evaluate how well question 1.4, developer effort, is fulfilled. In this case some effort is needed in order to separate the parts of the program in the first place, and to handle the errors once they occur. However, the developer does not need to concern himself with setting up the protection itself. Once the program is separated these issues are all taken care of by the system itself.

With either program crashing data loss is limited to the crashed module with the pipes acting as buffers for unprocessed data. In the two results listings, the sequence numbers from the processing element and the logger is of special interest. The processing will start at zero and increment by one for each message. If the processing process crashes and is restarted the sequence will start again from zero which can be seen in the first result. Meanwhile the sequence number for the logger continues as normal.

The opposite effect can be seen in the second result where the logging process is crashed and restarted. Its sequence number resets while the processing continues to run. Since the logger is nor restarted immediately the effect of the pipe buffering can be seen: the first logger messages after the crash contain multiple processing messages and none of its messages has been lost.

By observing the return value from `waitpid` the crashes can be handled in an appropriate manner. In this case the programs are simply restarted, however more complex systems may perform additional actions such as sending a message to an administrator, cleaning up resources or similar. By observing the timing of messages it is also possible for sibling programs to detect anomalies, such as the logger observing that messages are missing with the `no work` message.

## 8.5. Memory Copy Speed Test

In this section a simple speed test is performed to get a benchmark of how the solutions affect a basic program operation, memory copying. This is a runtime test affecting research question 2.2.

### 8.5.1. Methodology

The test is made out of a simple memory copy operation and recording how many times the memory was copied in a given time. This test is then run in three different contexts: Once before the memory protection is put to use, once as a regular process with the data in the `.bss` section, and once with the data in a dynamically allocated memory. The copy function is listed in listing .

```
1 void TestMemoryCopy (volatile unsigned int *data, unsigned int size,
2     unsigned int milliseconds)
3 {
4     unsigned int start = PlatformGetTick ();
5     unsigned int now = start;
6     unsigned int iterations = 0;
7
8     while ((now - start) < milliseconds)
9     {
10        unsigned int s = size / 2;
11
12        for (unsigned int i = 0; i < s; i += 8)
13        {
14            data[s + i + 0] = data[i + 0];
15            data[s + i + 1] = data[i + 1];
16            data[s + i + 2] = data[i + 2];
17            data[s + i + 3] = data[i + 3];
18            data[s + i + 4] = data[i + 4];
19            data[s + i + 5] = data[i + 5];
20            data[s + i + 6] = data[i + 6];
21            data[s + i + 7] = data[i + 7];
22        }
23
24        now = PlatformGetTick ();
25        iterations++;
26    }
27
28    Print ("Bandwidth: %i Bps\n", (1024 * 8 * iterations) / (milliseconds / 1000));
29 }
```

Listing 15: The C copy function

### 8.5.2. Expected Results

It is expected that the first case is the fastest operating one since it avoids the overhead of the scheduler, and does not suffer from interference with other programs. The next

cases are expected to perform about the same as each other, but slightly below the first case.

### 8.5.3. Test Execution

The tests are run for about five seconds each with a buffer of exactly 16 kilobytes.

- Before memory protection: `179.488358 MB/s`
- As a regular program, data in `.bss`: `141.172736 MB/s`
- As a regular program, data explicitly allocated: `130.020147 MB/s`

Next the tests are run for about 20 seconds each with the same buffer size.

- Before memory protection: `179.487539 MB/s`
- As a regular program, data in `.bss`: `141.196083 MB/s`
- As a regular program, data explicitly allocated: `130.021376 MB/s`

### 8.5.4. Discussion and Conclusion

As can be seen in the results, copy speed is fairly consistent. With the data in the `.bss` it performs at about 79% of the baseline performance. With data dynamically allocated using `mmap` the result is about 72% of the baseline performance.

These results are about as expected. The difference between the baseline and the protection-enabled results may seem large at first, however this is most likely due to the `PlatformGetTick` being a 32-bit memory fetch in the baseline and a full system call in the others. As system calls are naturally expensive it is to be expected that if the frequency of calls to `PlatformGetTick` were reduced, the performance gap would tighten.

Furthermore there are also background processes running during the regular program tests that may use some processing time and lower the results. The most obvious program is the shell that was used to start those tests, but there could also be worker threads running in the kernel and so on.

## 8.6. Added Context Switching Overhead

As shown in section 6.6, modification to the context switch is necessary in order to reprogram the MPU registers when the CPU is to switch between programs. If this is not done then the program that is given CPU time will have access to the wrong data, not even its own, and malfunction.

This reprogramming causes some extra time to be spent in the context switching mechanism. Since the operating system currently supports two different processors, the Cortex-M and the Cortex-R, there are two variants of the context switch. The part

of the context switch responsible for reprogramming the Cortex-M is shown in listing 9, and is remarkable efficient due to the MPU registers being memory mapped. Only eight instructions in total are needed for the reprogramming, however two of these are multi-word load and store instructions so the instruction count is not analogous to the number of CPU cycles.

The context switch of the Cortex-R, however, is a different story. This CPU variant does not have memory mapped registers for the MPU. Instead, each register must be programmed using the `mcr` instructions, which moves the contents of a normal register into a special register. Because there are six regions to reprogram with three registers each, four when including the region number, the instruction must be used for a total of 24 times. The total instruction count for the Cortex-R MPU reprogramming is 38 instructions, and the complete context switch is lised in appendix B.6.

Note that with no multi-word store instructions this is not the same as the number of clock cycles used. In other words, a much more in-depth analysis is required in order to get the actual amount of time used, since instructions have different execution times and may even have dependencies.

Note that the CPU that the solution is tested on runs at 168 MHz, which translates to $168,000$ instructions between each time the context switch is triggered naturally by the system timer, which operates at 1 KHz.

## 8.7. System Call Interface Overhead

As a consequence of the strict memory protection employed in the system, programs are not able to execute kernel code, which provides services and communications. How will they then communicate with the outside world and each other? And since the programs are dynamically loaded, how do they even know where the kernel functions are?

The purpose of this section is to give a general impression of how system calls affect run-time performance, relevant to research question 2.2.

### 8.7.1. System Calls

The answer is system calls using software triggered interrupts. When a program needs the kernel to perform a service for it, the command arguments are placed in the first CPU registers, called the argument registers `r0 - r3`. It then executes a special instruction, the ARM `svc` service call instruction. This instruction triggers an interrupt similar to those made by peripherals and other hardware. The CPU starts executing a handler for the interrupt which then performs the requested service on behalf of the program.

Since the system call instruction is a hardware mechanism, the user program does not need to know where the handler code actually is. The hardware takes care of the address of the handler function, and puts the CPU into a privileged state where it may access kernel code that is otherwise hidden. Since the hardware protects this mechanism it is not possible for user programs to specify arbitrary functions to execute, the interrupt always causes the same handler to execute.

### 8.7.2. Benefits and Limitations

The main benefit of using a system call interface is that the requesting program does not need to know where or how the service is implemented. From the perspective of the program, it just executed a magic instruction that did what it wanted, such as writing to a file or updating the time shown on a clock display. Due to the very nature of the system call interface, it is well specified, providing a clear set of kernel interfaces that the programmer may use. Since the program is not given direct access to the kernel code, security is also well protected when using a properly designed system call interface.

These benefits come at a cost however, which is performance. Making the service call in the first place takes time due to the extra work of finding the correct handler functions and verifying the arguments. The operating system cannot trust that the arguments given by the user program is in fact valid. Verifying these arguments takes time, especially when the arguments are memory arrays and string data of unknown length.

It is, however, few if any ways around a software interrupt system call interface when strict protection is desired.

### 8.7.3. Performance Study

In order to have a look at how system calls affect performance, we will look at how the `write` call is implemented, all the way from the user program to a driver handler. We choose a driver handler because interacting with drivers directly is a typical use case for embedded applications. In this case we will look at the `/dev/null` device driver, however, it could be any other driver.

#### User Program Parts

At first, we need to look at how the `write` call looks. The function prototype looks like this:

```
1 int write (int fd, void *buffer, size_t length);
```

What the read call does is to write the contents of `buffer`, with length `length`, into whatever the file descriptor `fd` represents. In our case, `fd` is a file descriptor opened for the `/dev/null` character device driver. Once completed, the function returns the number of bytes written or $-1$ for errors.

#### The C Standard Library

The C standard library is the underlying library of supporting functions for the C programming language, made up of commonly used functions. `write` is one of these functions. The job of the library is to hide the implementation of the call so that the end programmer does not need to concern himself of how the standard functionality works.

In this case, all the library needs to do is to trigger the software interrupt that was previously discussed. Listing 16 shows how the call is made behind the scenes. By

the time it is called, the compiler will already have placed the `fd`, `buffer`, and `length` arguments into the three first registers. A pointer to the global error value is written to the fourth register before the service call instruction is executed. Note that in this case the call number is 3. This number comes from the place in the system call table where the write call is placed.

```
1 .section .text._write
2 .global _write
3 .type _write, %function
4 _write:
5     ldr r3, =errno
6     svc 3
7     bx lr
8     nop
9 .size _write, .-_write
```

Listing 16: C Library implementation of the `write` function

### Interrupt Handler

When the interrupt is made, the CPU jumps to a pre-defined handler. This handler is listed in listing 17. Its job is to stack registers that will be overwritten, extract the system call number from the user `svc` instruction and use that to reference the table of system call function handlers. It then calls the handler using the correct arguments, and returns with the proper return value.

```
1  .weak SVCHandler
2  .type SVCHandler, %function
3  SVCHandler:
4      stmdb sp!, {r0-r3,r12,lr}    @ Stack regs that will be overwritten
5
6      ldrb r12, [lr, #-1]          @ Load argument to SVC
7      and r12, r12, #0x3F          @ Make sure to keep index within jump table bounds
8      ldr lr, =svcJumpTable        @ Load address of jump table
9      ldr r12, [lr, r12, lsl #2]   @ Load entry by shifting argument two times
10
11     ldr r0, =currentthc          @ Load address of current thread
12     ldr r0, [r0]                 @ Load the actual pointer
13     mov r1, sp                   @ Store pointer to args
14
15     blx r12                      @ Branch and link to entry
16     str r0, [sp]                 @ Store return value
17
18     ldmia sp!, {r0-r3,r12,lr}    @ Unstack
19     movs pc, lr                  @ Return, no subs for svc handler
20 .size SVCHandler, .-SVCHandler
```

Listing 17: Assembly handler for the `svc` instruction

As can be seen from listing 17 there is more to the procedure than a single function call. It has a total of 12 instructions, of which two are multi-word stores and loads.

### Write System Call Handler

Next up is the first handler function that handles the incoming write call. This function is rather straight forward: get the actual data from the file descriptor, `fd`, from the internal data and call the virtual file system `vfs_write` function. The handler function is listed in listing 18.

```
int svc_write (struct ThreadControl *ctc, void *args)
{
    // File descriptor index from the caller
    unsigned int pfd = ((unsigned int*)args)[0];

    // Index to file descriptor in the kernel
    int kfd;

    // Check for validity and get kfd
    if (pfd < MAX_NODES && (kfd = ctc->pctrl->fnodes[pfd]) != 255)
    {
        int ret = vfs_write (&nodetable[kfd], (char*)((int*)args)[1], ((int*)args)[2]);

        if (ret < 0)
        {
            *ERRN = -ret;
            return -1;
        }

        return ret;
    }

    // Invalid descriptor
    *ERRN = EBADF;
    return -1;
}
```

Listing 18: Kernel mode write call handler

The real contents of the file descriptor is hidden from the user program to make the user part of the system call easier, less error prone and more secure. As a consequence the data must be fetched from the current execution context, and the file descriptor must be validated in the first place. This clearly has some overhead, but due to how the kernel data structures are laid out it is done in constant time. Note that the handler also has to set the error value correctly if required.

### Virtual File System and The Device File System

A common concept in the world of operating systems is that of a virtual file system. Instead of having a dedicated file system and interface for each device, or file system format, all devices are located in one tree structure. This is also known as the "everything is a file" concept in the world of *NIX systems.

The virtual file system is responsible for calling the write handler that belongs to the correct file system. In this case, we have opened a file descriptor to a device driver that is part of the `/dev` file system, and so the implementation of `vfs_write` does nothing but call the correct write function through a file system pointer that has been set when the descriptor was initially opened. The `vfs_write` function is listed in listing 19.

```
1 int vfs_write (struct FileNode *fnode, char *data, int n)
2 {
3     return fnode->fs->write (fnode, data, n);
4 }
```

Listing 19: Virtual file system write call handler

As made clear in listing 19 this function is rather simple, all it does is to load the file system pointer, which contains a list of functions, and call write for that file system.

Because the device file system really is nothing but a list of devices it is equally simple, all it has to do is to call write for the correct driver. This is also done through function pointers, and can be seen in listing 20.

```
1 int vfs_dev_write (struct FileNode *node, void *data, int nbytes)
2 {
3     return node->dev->write (node, data, nbytes);
4 }
```

Listing 20: Device file system write call handler

### The Device Driver

At last, the execution ends up in the device driver. In our case this is `/dev/null`, which plays a special role in the UNIX environment: it acts as a black hole for unwanted data. Programmers and users alike typically use it to redirect unwanted data, the contents is ignored and the driver acts like it accepted it. Therefore, the implementation is really simple, all it needs to do is to return the number of bytes that was supplied. The implementation is shown in listing 21.

```
1 int dev_null_write (struct FileNode *node, void *data, unsigned int nbytes)
2 {
3     return nbytes;
4 }
```

Listing 21: Write function of `/dev/null`

The `/dev/null` device driver is by far the simplest type of driver found in a system: it does absolutely nothing. It does, however, illustrate how calls from the user program to the driver is made. Other drivers are far more complex. For instance, the implementation of the serial interface write function is over 100 lines excluding any helper functions. Actual file systems that work with on-disk data structures are far more complex and easily stretches beyond thousands of lines.

### 8.7.4. Summary

The previous sections describe how a system call is made, complete from the user code to a driver. A total of six functions are involved along the way of varying complexity. The performance impact can be found in mainly two places:

1. The interrupt handler. Not just a function call, the interrupt handler has to stack and set registers properly, fetch and verify the system call number, make the actual call and handle returning back to user code.

2. The first write handler. It has to get the actual data structure for the file descriptor, and must verify that the descriptor is valid while doing so. It must also handle error values.

In the case of the file systems, there is little to mention in this case. Once a reference to a device driver is opened, all that needs to be done is to get the correct write function and call it.

This is not necessarily the same for other file systems that involve actual files. However, with the current design, only the relevant code is executed due to the excessive use of function pointers.

The use of system calls are easily much more expensive than just a function call, as can be seen from the previous section. Regular system calls are usually just a branch-and-link instruction with some argument stacking by comparison. Sadly, the use of system calls are more or less required for strict protection schemes. Care must be taken in order to avoid excessive calls. Examples of methods that may reduce the number of calls are:

- Shared memory buffers between programs and the kernel. Once the buffers are agreed upon, data may quickly be moved.

- Batch processing. Instead of making one system call per item that needs processing, items are grouped and processed together.

- User buffering. Fewer system calls are necessary if the user program reads or writes in larger blocks and then operates on this buffer in smaller portions internally.

# 9. Discussion

In this chapter a discussion of the thesis as a whole is performed.

## 9.1. Solution Discussion

This section provides a discussion of the solution chapters, including the functionality, performance and usability tests done of the implemented solutions.

### 9.1.1. Memory Allocation

The memory allocation subsystem has proven itself as functional and usable for allocating protected process memory, with additional memory available upon request. As discussed in section 8.1 there are however some room for performance improvements that may be useful when an application does frequent allocations and de-allocations through the `mmap` API. These improvements would make the allocations quicker by enhancing the allocation algorithms, and the page misses cheaper by improving the time complexity of the page walks.

In order to increase the confidence in the allocation algorithm implementation it would be beneficial to perform better, more in-depth automated testing.

A formal in-depth code review would also be beneficial in order to uncover weaknesses in the implementation that could lead to sub-optimal allocations and usage of memory.

### 9.1.2. Program Loading

The program loading mechanism presented in chapter 6 has as with the allocation subsystem proven itself as fully functional. Following the POSIX standard, it has allowed programs to be compiled for desktop and embedded systems alike with no source modifications.

Although fully functional and very useful indeed, the program loading subsystem is the most complex and time consuming of the subsystems proposed. This is due to the disk accesses that must be performed involving slow storage devices, all the memory setup, and the relocation processing that must be performed on the binary before its execution can start.

If the loading is all done in the `posix_spawn` system call handler, other applications will be blocked completely until the loading is finished. Depending on the complexity of the program that is to be loaded and the performance of the storage device where the file is located, the operation may take from a fraction of a second to many full seconds to complete. This has a disastrous effect on other applications, especially real-time

ones such as audio applications. The performance impact has been eased for real-time applications by executing the work in a separate worker thread. It is then possible to allow background work to continue with minimal interference. The operation of loading in itself is however not made faster by this. Most of the time is spent resolving relocation entries which refers two table in the file, which again causes excessive seeking. Other than an elaborate disk caching scheme, general optimization of the implementation may provide a performance benefit with continued development.

### 9.1.3. Paging Mechanism

The paging mechanism provides a way for programs to allocate additional memory after they have been started. Most embedded software does not use excessive amounts of free memory, and so the implementation has been focused on functionality and simplicity instead if performance. It is however able to handle a small set of allocations without severe performance penalties. During benchmarking that is described in section 8.5, the penalty is shown to be 7% compared to memory located in the `.bss` section.

The paging mechanism is intended for the cases where static allocation is not appropriate, due to not knowing how much memory is needed at compile time. An example is a text editor that is designed to work with files of different sizes. With static memory allocation, a static buffer must be allocated at all times that is large enough for the largest file, which would cause unacceptable memory usage and inefficiency. The pager enables the use of the `mmap` API for dynamic allocation, solving these cases.

## 9.2. Research Question Discussion

In this section the results of the testing is discussed with respect to the research questions, in order to get an impression of how well the goals have been fulfilled.

### 9.2.1. Research Question 1

#### Software Fault Tolerance

Research question 1.1 is about how different processes or programs should not be able to harm each other or the system itself by a bad memory reference. The solutions presented and tested earlier all build up to a system where each process, or program, are all separated from each other using the memory protection hardware. Separation refers to the practice of restricting the set of memory locations that a process may access to those that it owns itself.

As shown in sections 8.3 and 8.4 the system is indeed able to detect and handle occurrences of bad memory accesses appropriately. The offending program is stopped immediately and its owner is notified, which may then act accordingly.

#### Security

Research question 1.2 however can only partially be fulfilled since security bugs may exist regardless of memory protection in itself. With the separation of process memory areas any remaining bugs will in any case be limited to their own process, the subsystem they are part of or exploits in the kernel itself. The attack vectors are limited, not just because of process separation but also due to the enforcement of access rights, such as executable regions also being read-only.

#### Hardware Error Tolerance

Similarly the occurrence of hardware error will trigger events that are similar to those described in both research question 1.1 and research question 1.2. Since hardware error is likely to occur in the form of random memory corruption however it can be assumed that most will be detected by either the memory protection or the CPU itself as bad memory references or bad instructions. If the corruption occur in the kernel, however, the error is likely to be fatal. Research question 1.3 can, therefore, only be partially fulfilled for the same reason as to why question 1.2 is partially fulfilled.

In any case the system is much more likely to detect errors and be able to continue operating or alert a user. This increases the perceived reliability and robustness of the application. As for security in general, the effort required to breach the system is effectively increased, however it would require a more in-depth analysis to determine exactly by how much.

#### Ease of Use

As described in section 6.4, a standard and known API has been selected in order to let programmers use existing knowledge and code, simplifying the task of writing software significantly. The selected API is `posix_spawn`, which is a variant intended for simpler systems. This API handles the loading of programs into memory, and uses the memory allocation and paging mechanisms to implement the strictly non-shared memory architecture.

How this works in practice is shown in section 8.4 with the small system use case: as long as the programmer finds a way to split the larger application into smaller components, memory protection is applied by the system automatically. The programmer does not have to care about how the protection actually works or how to use it, as is the case with other microcontroller systems. With this it can be assumed that research question 1.4 is satisfied, since the protection is used without added developer effort.

### 9.2.2. Research Question 2

As made clear during testing, the performance impact of the solution(s) depend on the end application and its design. While some performance limitations may effectively be solved others are inherent to dynamic loading. Taking these limitations into account

when designing the application, it is possible to design a system where the performance impact is negligible.

This is especially important for research question 2.2 due to the very nature of real-time performance. Applications that depend on real-time characteristics may need to move critical code into the kernel where many limitations are lifted, and use protected processes only for less real-time constrained code. A typical example of such architecture is the relationship of hardware drivers and its hardware on the real-time dependent side and the user of the driver on the non-real-time dependent side.

The memory copy test shows a performance impact of 21 % compared to running the test bare-metal. The test is flawed, however: it does not separate the performance impact of the system call interface from the impact of the memory protection system itself. When the tested memory is moved from the default `.bss` section to memory allocated using the pager, an additional 7 % decrease in performance is experienced.

A discussion on other run-time performance problems are then performed regarding the context switching mechanism and the system call interface. The context switch uniformly affects performance, and would be fairly constant since context switches happen at a regular interval. The number of instructions are counted and analyzed. The total instruction count for reprogramming the MPU on the Cortex-R MPU is counted to 38 instructions, and the complete context switch is lised in appendix B.6.

The system call interface is analyzed by looking at how a commonly used call is performed. It gives an impression on how and why a system call will affect performance, and how such calls should be used to limit the impact.

As for research question 2.1, there is currently no known competitor to the presented solution other than going back to not using memory protection altogether or using it manually. As long as programs are developed with separation and loading in mind there is no extra effort involved, and run-time performance must be judged on a per-application basis since different programs will be affected in different ways. A summary on how the performance effects of system calls may be controlled is summarized in section 8.7.4.

# 10. Conclusion

In this thesis, a system of memory allocation, program loading and protection mechanisms have been presented that lets a developer to experience the full convenience of desktop application programming for smaller embedded systems. Through a standardized POSIX API applications may be compiled and tested for both systems without modification.

Strong memory separation is achieved by applying hardware protection for each separate program region and making their access mutually exclusive in-between programs. Communication is forced to be performed through “official” channels such as fifos, pipes, sockets and files ensuring that programs remain separate while still allowing communication.

Whenever a program experiences a crash or fault it is then possible to contain the damages to the faulting process with high probability. This makes it possible for other programs to retain their state and possibly continue working. A fault in one process is reported to the owner process that started it allowing the error to be handled. The application programmer may then choose how to handle the error. Likely options are, for instance, restarting the faulted program or performing a graceful restart of the entire system with minimal data loss.

In the benchmark performed in section 8.5, it is shown that the performance impact of the solution can be as high as 28% depending on the application characteristics. The performance impact is expected to be improved with continued development as many of the systems have room for improvement.

# 11. Future Work

This thesis has presented an easy to use solution for strict memory protection in deeply embedded systems where cost is of importance. Although it works well, there are always room for improvements.

## 11.1. Memory Allocation Improvements

Currently the memory allocator does not take all possibilities into account when a block is allocated. For instance, it does not investigate using two regions to protect one block that spans the boundary of an alignment address, or to use a larger region to satisfy alignment constraints. In its current form, these issues may lead to less than optimal allocation that uses more memory than strictly necessary. Solving the issues may allow an application to use smaller and cheaper hardware in the end product at the cost of more complicated code.

Another issue that remains to be solved is how to deal with having multiple memory banks. Some microcontrollers have more than one RAM module. These often have different characteristics such as size, speed, Direct Memory Access (DMA) capabilities etc. A good API must be provided for programs that desire specific characteristics, and a good heuristic for selecting a memory for the programs that do not, must be implemented.

## 11.2. Paging Improvements

The current paging mechanism uses a global linear list for the page table. It is trivial to implement and has great performance when a low amount of pages is needed. Unfortunately it scales poorly with the number of pages since the search time is linear. The two major paths of improvements are splitting the table into per-process tables and designing a tree structure for storing the data.

With these improvements the performance impact of multiple small allocations would not be as great as it is currently. Since most embedded software already makes scarce use of dynamic memory however, other subsystem improvements may yield better cost to benefit ratios.

## 11.3. Memory Sharing Schemes

Currently there are four main ways of sharing data between processes implemented: pipes, sockets, regular files and device drivers. The pager opens up a fifth option: shared

memory between processes, which is explicitly agreed upon. Such an option would be the highest bandwidth option for communication. This is currently not implemented, however it should be rather straight forward to do so since the API is fairly mature for desktop systems.

## 11.4. Improved Standards Compliance

Most of the APIs only cover the most basic use cases. Adding support for more features would make it easier to adapt software that already makes use of these, not having to modify software libraries. This especially true for the standard C library `libcglue` which currently only implements the basics and relies upon `newlib` to fill in the rest. Improving `libcglue` to the point where `newlib` can be removed would improve system compatibility and resource usage since `newlib` is rather large.

## 11.5. Hardware Error Handling

The hardware used for the satellite computer was selected specifically for its many error handling and prevention features. These features are not fully utilized in the current software. Improving the handling of detectable hardware errors would improve reliability of the software even further.

## 11.6. Syscall Data Access Checking

In order to make the memory protection "bullet proof" all system calls must be validated so that the kernel does not violate the protection on behalf of the program. These checks come in addition to the previously described mechanisms and has not yet been implemented. While the previous methods protect against accidental errors, system call checks must be done in order to secure against intentional security attacks when the attacker is allowed to run arbitrary programs.

## 11.7. Moving Data Back Into Flash

By default, the operating system loads both code and data into main RAM. While this is extremely convenient during development, it increases the RAM demands by a large amount while leaving the microcontroller flash largely unused.

It is beneficial to place the code sections in the internal flash for programs deployed in the field without losing any of the protection capabilities provided by the operating system. A possible solution to this problem is to create an Execute In Place File System (XIPFS), allowing code to run from flash without any change to the execution environment, in turn allowing the use of smaller and cheaper microcontrollers. During development the file system image must be built and flashed alongside the kernel. The saving in RAM usage would come at the cost of an extra step during development.

## 11.8. Test Procedures

The operating system and its core components is currently tested using the typical open source method: use it until it breaks. While this is often good enough for general applications it is not for mission-critical satellite systems. Unit and integration testing should be done for important kernel components so that bugs, limitations and regressions are kept to a minimum. Preferably these tests should be automated, see 11.9.

## 11.9. Build Automation

In combination with test procedures, build automation should be used to run the tests automatically every time anything is changed and pushed to the software repositories. Modern development tools have extensive integration for such tools, making it possible to require that code is proper and sound before it is accepted. When testing procedures are set up and written build automation should also be considered to improve the product.

## 11.10. Power Management

Since the main applications for the software is embedded systems that are often battery powered, power management is important. This should be improved before deploying in any battery powered system.

## 11.11. Kernel Security

The kernel does currently not support users and groups for access control. For more complicated applications, especially those that desire to run non-verified user code as plug-ins, access control would be a desired feature to lock down what a program may do.

## 11.12. General Operating System Development

At last but not least there is general development of the operating system. This general development encompasses new feature development, maintenance and driver development.

# References

[1] Page Table Management. https://www.kernel.org/doc/gorman/html/understand/understand006.html, 2002.

[2] OpenGroup mmap documentation. https://pubs.opengroup.org/onlinepubs/007904875/functions/mmap.html, 2004.

[3] OpenGroup munmap documentation. https://pubs.opengroup.org/onlinepubs/007904875/functions/munmap.html, 2004.

[4] The Open Group Base Specifications Issue 6. https://pubs.opengroup.org/onlinepubs/009695399/basedefs/contents.html, 2004.

[5] Tanenbaum, Andrew S. *Modern Operating Systems. 3rd ed.* Pearson/Prentice Hall, 2009.

[6] NORD-1 Reference Manual. http://sintran.com/sintran/library/libhw/NORD-1-RM-1-EN.pdf, Feb 1970.

[7] Xilinx Zynq UltraScale+ MPSoC product advantages. https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html, Nov 2018.

[8] MCU Market on Migration Path to 32-bit and ARM-based Devices. http://www.icinsights.com/news/bulletins/MCU-Market-On-Migration-Path-To-32bit-And-ARMbased-Devices/, Apr 2013.

[9] Cortex-M4 Devices Generic User Guide. http://infocenter.arm.com/help/topic/com.arm.doc.dui0553a/DUI0553A_cortex_m4_dgug.pdf, Dec 2010.

[10] Cortex-R4 and Cortex-R4F Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0363g/DDI0363G_cortex_r4_r1p4_trm.pdf, Apr 2011.

[11] David Diamond Linus Torvalds. *Just for Fun.* Harper Business, 2001.

[12] Kumar, Ram, Akhilesh Singhania, Andrew Castner, Eddie Kohler, and Mani Srivastava. *A System for Coarse Grained Memory Protection in Tiny Embedded Processors.* Proceedings of the 44th Annual Design Automation Conference, 2007.

[13] Lopriore, Lanfranco. *Memory Protection in Embedded Systems.* Journal of Systems Architecture 63, 2016.

[14] Suzuki, S., and K.G Shin. *On Memory Protection in Real-Time OS for Small Embedded Systems.* Proceedings Fourth International Workshop on Real-Time Computing Systems and Applications, 1997.

[15] 33C3: Hunz Deconstructs the Amazon Dash Button. https://hackaday.com/2017/02/02/33c3-hunz-deconstructs-the-amazon-dash-button/, Feb 2017.

[16] ARM Strategic Report 2015. http://www.annualreports.com/Company/arm-holdings-plc, 2015.

[17] Arm Holdings Q1 2017 Roadshow Slides. http://www.arm.com/-/media/global/company/investors/pdfs/arm_sb_q3_2017_roadshow_slides_final.pdf, 2017.

[18] ELF for the ARM©Architecture. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0044f/IHI0044F_aaelf.pdf, Nov 2015.

[19] ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition. https://static.docs.arm.com/ddi0406/cd/DDI0406C_d_armv7ar_arm.pdf, Mar 2018.

[20] OpenGroup `sys/mman.h`. https://pubs.opengroup.org/onlinepubs/007904875/basedefs/sys/mman.h.html, 2004.

[21] `mmap`(2) Manual Page. https://linux.die.net/man/2/mmap, 2018.

# Appendices

# A. Supplied Archive

The code of the implementation is supplied in an archive with this thesis. Only the most relevant code is supplied in this appendix. Note that in order to run most of the software, appropriate hardware must be used, available from the Orbit satellite group.

The archived code must be considered a frozen image of an on-going development effort. For anything other than historical references, the current software repositories are recommended: `https://git.erlendjs.no/erlends-os` for the operating system itself and `https://git.orbitntnu.no/orbit` for the satellite. **The archived code supplied with this thesis should not be used and redistributed, the repositories linked above should be used instead.** This is because it is guaranteed to be outdated. Some software components are omitted from the archive, that are not relevant to the understanding of this thesis.

## A.1. Supplied Archive Structure

The supplied archive contains a partial frozen version of the main repository used for the Orbit satellite group. The most important parts can be found in these locations:

- `/README.md`
  A copy of the text found here, written in markdown.
- `/orbit-main-repository/README.md`
  The main readme for the satellite group. It contains a quick start guide on how to download, compile, run and use the operating system and the programs that run on it. The "live" version of this readme can be found at `https://git.orbitntnu.no/orbit/orbit`, which should be your main entry point to the project.
- `/orbit-main-repository/bin`
  This directory contains extremely simple versions of your typical UNIX core utilities, such as the file copy program `cp` and a shell `sh`. These are all regular programs that make full use of the solutions presented in this thesis, and most can be used as code examples.
- `/orbit-main-repository/docs`
  Contains the wiki pages that adds to the main repository readme. It describes various subjects in greater detail.
- `/orbit-main-repository/kernel/kernel`
  This is the kernel repository. All the code for the solutions presented in this

thesis can be found here. Headers are located in `inc` and sources are located in `src`. The directory naming should be mostly self explanatory, however, you may find the most relevant parts of the thesis in the following locations: memory allocation in `mem` and `mpu`, process loading in `process` and `elf`, and at last paging in `mpu/pager.c`.

- `/orbit-main-repository/kmod`
  Kernel modules are kept here. At the moment there is only one, the `fatfs` file system module, which is the only third party developed part of the kernel. Note that the code of the file system itself is not supplied in this archive, and must be supplied separately.

- `/orbit-main-repository/lib`
  This directory contains libraries. The most important is the `libcglue` library, which is the standard C library. More libraries can be found at `git.erlendjs.no` and `git.orbitntnu.no`.

- `/orbit-main-repository/programs`
  This directory contains user programs specifically written for the satellite. Since these are not really related to the thesis and contains second and third party code, only the test programs are supplied. Visit `git.orbitntnu.no` for the actual satellite programs.

- `/orbit-main-repository/usr`
  Contains all other programs that are not important core utilities. Similar to the other directories, this also only contains the most important ones. The full collection of 24 (at the time of writing) user programs can be found at `git.erlendjs.no`.

- `/orbit-main-repository/util`
  This directory contains some generalized development utilities, such as parts of the build system, file transfer programs and repository management.

## A.2. Code Licensing

The kernel is licensed under the GNU General Public Licence (GPL) Version 3. See the licence in the `/orbit-main-repository/kernel/kernel/copying` file for more details.

Other programs are mostly licensed under the BSD license, while some are under GPL. See their appropriate directories and files for more information.

# B. Code Listings

## B.1. Memory Allocation Test Code

```
1 void PlatformTestMem (void)
2 {
3     // Step 1
4     unsigned int a = MemAlloc (1000, 0);
5     Print ("Step 1: Address 0x%x\n", a);
6 
7     // Step 2
8     unsigned int b = MemAlloc (1000, 0);
9     Print ("Step 2: Address 0x%x\n", b);
10 
11    // Step 3
12    struct MemMpuReturn mpu;
13    unsigned int c = MemAllocMpu (7000, 0, &mpu);
14    Print ("Step 3: Address 0x%x\n   dataptr 0x%x\n   rbar 0x%x\n   rasr 0x%x\n",
15        c, mpu.dataptr, mpu.mpu_rbar, mpu.mpu_rasr);
16 
17    // Step 4
18    MemFree (b);
19    Print ("Step 4: Done\n");
20 
21    // Step 5
22    unsigned int d = MemAllocMpu (512, 0, &mpu);
23    Print ("Step 5: Address 0x%x\n   dataptr 0x%x\n   rbar 0x%x\n   rasr 0x%x\n",
24        d, mpu.dataptr, mpu.mpu_rbar, mpu.mpu_rasr);
25 
26    // Step 6
27    unsigned int e = MemAlloc (200, 0);
28    Print ("Step 6: Address 0x%x\n", e);
29 
30    // Step 7
31    MemFree (a);
32    Print ("Step 7: Done\n");
33 
34    // Step 8
35    MemFree (c);
36    Print ("Step 8: Done\n");
37 
38    // Step 9
39    MemFree (d);
40    Print ("Step 9: Done\n");
41 
42    // Step 10
43    MemFree (e);
```

```
44     Print ("Step 10: Done\n");
45 }
```

## B.2. Shell Function For Program Execution

```
1 /*
2  * Start a process, putting the resulting pid in *pid.
3  * The infile and outfile arguments are optional. If supplied,
4  * they specify the name of a file that will replace stdin or stdout.
5  * The last argument outappend specifies if the output file is opened
6  * in append mode or not.
7  */
8 int CommandStartProcess (pid_t *pid, char *infile, char *outfile, int outappend)
9 {
10     // Find executable file with first argument
11     char *name = CommandFindProcessImage (pargs[0]);
12
13     // Check if we got anything
14     if (!name)
15     {
16         errno = ENOENT;
17         return 1;
18     }
19
20     // Spawn attributes object
21     posix_spawnattr_t attr;
22
23     // Initialize it
24     if (posix_spawnattr_init (&attr))
25         SimplePrint ("attr_init failed\n");
26
27     // Set process group
28     if (posix_spawnattr_setflags(&attr, POSIX_SPAWN_SETPGROUP))
29         SimplePrint ("attr_setflags failed\n");
30
31     if (posix_spawnattr_setpgroup (&attr, 0))
32         SimplePrint ("attr_setpgroup failed\n");
33
34     // Prepare for changing input and output file
35     posix_spawn_file_actions_t factions;
36
37     if (posix_spawn_file_actions_init (&factions))
38         SimplePrint ("factions init failed\n");
39
40     // If input file
41     if (infile)
42     {
43         if (posix_spawn_file_actions_addopen (&factions, 0, infile,
44             O_RDONLY, S_IRWXU | S_IRWXG | S_IRWXO))
45             SimplePrint ("addopen in failed\n");
46     }
47
48     // If output file
```

```
49     if (outfile)
50     {
51         int flags = O_WRONLY | O_CREAT;
52 
53         flags |= outappend ? O_APPEND : O_TRUNC;
54 
55         if (posix_spawn_file_actions_addopen (&factions, 1, outfile, flags,
56             S_IRWXU | S_IRWXG | S_IRWXO))
57             SimplePrint ("addopen out failed\n");
58     }
59 
60     // Launch process!
61     int ret = posix_spawn (pid, name, &factions, &attr, pargs, environ);
62 
63     // Delete attr
64     if (posix_spawnattr_destroy (&attr))
65         SimplePrint ("attr destroy failed\n");
66 
67     // Delete file actions
68     if (posix_spawn_file_actions_destroy (&factions))
69         SimplePrint ("factions destroy failed\n");
70 
71     // Done
72     return ret;
73 }
```

## B.3. Crash Test Code

```
1 /*
2  * Copyright (c) 2018, Erlend Sveen
3  * All rights reserved.
4  *
5  * Redistribution and use in source and binary forms, with or without
6  * modification, are permitted provided that the following conditions are met:
7  *
8  * 1. Redistributions of source code must retain the above copyright notice, this
9  *    list of conditions and the following disclaimer.
10 * 2. Redistributions in binary form must reproduce the above copyright notice,
11 *    this list of conditions and the following disclaimer in the documentation
12 *    and/or other materials provided with the distribution.
13 *
14 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
15 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
16 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
17 * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR
18 * ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
19 * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
20 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
21 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
22 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
23 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
24 */
25 
```

```
26 ////////////////////////////////////////////////////////////////////////////////
27 // Includes
28 ////////////////////////////////////////////////////////////////////////////////
29 #include <stdio.h>
30 #include <string.h>
31 #include <unistd.h>
32 #include <fcntl.h>
33 #include <errno.h>
34 #include <sys/ioctl.h>
35 #include <sys/mman.h>
36
37 ////////////////////////////////////////////////////////////////////////////////
38 // Functions
39 ////////////////////////////////////////////////////////////////////////////////
40 int nullptr (char **argv)
41 {
42     printf ("Testing access violation\n");
43
44     int a = 42;
45     volatile int *a1 = 0;
46     volatile int *a2 = &a;
47
48     if (*a1 == *a2)
49         printf ("Failed: Got equal\n");
50     else
51         printf ("Failed: Got unequal\n");
52
53     return 0;
54 }
55
56 int ptr (int argc, char **argv)
57 {
58     if (argc != 3 || strlen(argv[2]) != 8)
59     {
60         printf ("Invalid arguments, specify address in hex\n");
61         return 1;
62     }
63
64     int a = 42;
65     int b = 0;
66     volatile int *a1 = 0;
67     volatile int *a2 = &a;
68
69     for (int i = 0, j = 7; i < 8; i++, j--)
70     {
71         if (argv[2][i] >= '0' && argv[2][i] <= '9')
72             b |= (argv[2][i] - '0') << (j * 4);
73         if (argv[2][i] >= 'a' && argv[2][i] <= 'f')
74             b |= (argv[2][i] - 'a' + 10) << (j * 4);
75         if (argv[2][i] >= 'A' && argv[2][i] <= 'F')
76             b |= (argv[2][i] - 'A' + 10) << (j * 4);
77     }
78
79     a1 = (volatile int*) b;
```

```
80
81     printf ("Testing pointers\n");
82
83     if (*a1 == *a2)
84         printf ("Failed: Got equal\n");
85     else
86         printf ("Failed: Got unequal\n");
87
88     return 0;
89 }
90
91 int singlemmap (int argc, char **argv)
92 {
93     printf ("Allocating space: 1024 bytes\n");
94
95     int size = 1024;
96     int *addr = mmap (0, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
97
98     if (addr == MAP_FAILED)
99     {
100        printf ("Allocation failed\n");
101        return 1;
102    }
103
104    printf ("Interacting with memory...\n");
105
106    volatile int *a = addr;
107    a[0] = 10;
108    a[2] = 10;
109
110    if (a[0] != a[2])
111        printf ("Data was not stored\n");
112    if (a[1] != 0)
113        printf ("Data is not zeroed\n");
114
115    printf ("Unmapping\n");
116    munmap (addr, size);
117    return 0;
118 }
119
120 int doublemmap (int argc, char **argv)
121 {
122    printf ("Allocating space: 1024 bytes x2\n");
123
124    int size = 1024;
125    int *addr1 = mmap (0, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
126    int *addr2 = mmap (0, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
127
128    if (addr1 == MAP_FAILED || addr2 == MAP_FAILED)
129    {
130        printf ("Allocation failed\n");
131        return 1;
132    }
133
```

```
134     printf ("Interacting with memory...\n");
135
136     volatile int *a = addr1;
137     volatile int *b = addr2;
138
139     a[0] = 10;
140     a[2] = 10;
141
142     if (a[0] != a[2])
143         printf ("Data was not stored\n");
144     if (a[1] != 0)
145         printf ("Data is not zeroed\n");
146
147     memcpy ((void*)b, (void*)a, size);
148
149     if (b[0] != 10 || b[1] || b[2] != 10)
150         printf ("Data copy failed\n");
151
152     printf ("Unmapping\n");
153     munmap (addr1, size);
154     munmap (addr2, size);
155     return 0;
156 }
157
158 int mmapleak (int argc, char **argv)
159 {
160     printf ("Allocating space and leaking it: 1024 bytes\n");
161
162     int size = 1024;
163     mmap (0, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
164     return 0;
165 }
166
167 ////////////////////////////////////////////////////////////////////////////////
168 // Usage
169 ////////////////////////////////////////////////////////////////////////////////
170 int PrintUsage (void)
171 {
172     printf (
173         "Usage: crashtest [options]\n"
174         "  nullptr\n"
175         "    Test access to 0x0\n"
176         "  ptr <addr>\n"
177         "    Test access to addr\n"
178         "  singlemmap\n"
179         "    Test a simple mmap allocation\n"
180         "  doublemmap\n"
181         "    Test a double mmap allocation\n"
182         "  mmapleak\n"
183         "    Test leaking memory\n"
184         "  --help\n"
185         "    Print this message\n");
186     return 0;
187 }
```

```
188
189 int PrintUnknownCommand (void)
190 {
191     printf (
192         "Invalid argument, use\n"
193         "   crashtest --help\n"
194         "for a usage list\n");
195     return 1;
196 }
197
198 ////////////////////////////////////////////////////////////////////////////////
199 // Main function
200 ////////////////////////////////////////////////////////////////////////////////
201 int main (int argc, char **argv)
202 {
203     // If no arguments, print default
204     if (argc == 1)
205         return PrintUnknownCommand ();
206
207     // Check arguments
208     if (!strcmp (argv[1], "nullptr"))
209         return nullptr (argv);
210     else if (!strcmp (argv[1], "ptr"))
211         return ptr (argc, argv);
212     else if (!strcmp (argv[1], "singlemmap"))
213         return singlemmap (argc, argv);
214     else if (!strcmp (argv[1], "doublemmap"))
215         return doublemmap (argc, argv);
216     else if (!strcmp (argv[1], "mmapleak"))
217         return mmapleak (argc, argv);
218     else if (!strcmp(argv[1], "--help"))
219         return PrintUsage ();
220     else
221         return PrintUnknownCommand ();
222 }
```

## B.4. Example Use Case Test Code

```
1  /*
2   * Copyright (c) 2018, Erlend Sveen
3   * All rights reserved.
4   *
5   * Redistribution and use in source and binary forms, with or without
6   * modification, are permitted provided that the following conditions are met:
7   *
8   * 1. Redistributions of source code must retain the above copyright notice, this
9   *    list of conditions and the following disclaimer.
10  * 2. Redistributions in binary form must reproduce the above copyright notice,
11  *    this list of conditions and the following disclaimer in the documentation
12  *    and/or other materials provided with the distribution.
13  *
14  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
15  * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
```

```
16  * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
17  * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR
18  * ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
19  * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
20  * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
21  * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
22  * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
23  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
24  */
25
26 ////////////////////////////////////////////////////////////////////////////////
27 // Includes
28 ////////////////////////////////////////////////////////////////////////////////
29 #include <string.h>
30 #include <stdio.h>
31 #include <unistd.h>
32 #include <fcntl.h>
33 #include <errno.h>
34 #include <time.h>
35 #include <sys/ioctl.h>
36
37 #include <spawn.h>
38 #include <sys/wait.h>
39
40 ////////////////////////////////////////////////////////////////////////////////
41 // Externs
42 ////////////////////////////////////////////////////////////////////////////////
43 extern char **environ;
44
45 ////////////////////////////////////////////////////////////////////////////////
46 // Functions
47 ////////////////////////////////////////////////////////////////////////////////
48 /*
49  * Start a process, putting the resulting pid in *pid.
50  * The infile and outfile arguments are optional. If supplied,
51  * they specify the name of a file that will replace stdin or stdout.
52  * The last argument outappend specifies if the output file is opened
53  * in append mode or not.
54  */
55 int CommandStartProcess (pid_t *pid, char *name, char *infile, char *outfile,
56     int outappend, char **pargs)
57 {
58     // Spawn attributes object
59     posix_spawnattr_t attr;
60
61     // Initialize it
62     if (posix_spawnattr_init (&attr))
63         printf ("attr_init failed\n");
64
65     // Set process group
66     if (posix_spawnattr_setflags (&attr, POSIX_SPAWN_SETPGROUP))
67         printf ("attr_setflags failed\n");
68
69     if (posix_spawnattr_setpgroup (&attr, 0))
```

```
70          printf ("attr_setpgroup failed\n");
71
72      // Prepare for changing input and output file
73      posix_spawn_file_actions_t factions;
74
75      if (posix_spawn_file_actions_init (&factions))
76          printf ("factions init failed\n");
77
78      // If input file
79      if (infile)
80      {
81          if (posix_spawn_file_actions_addopen (&factions, 0, infile,
82              O_RDONLY, S_IRWXU | S_IRWXG | S_IRWXO))
83              printf ("addopen in failed\n");
84      }
85
86      // If output file
87      if (outfile)
88      {
89          int flags = O_WRONLY | O_CREAT;
90
91          flags |= outappend ? O_APPEND : O_TRUNC;
92
93          if (posix_spawn_file_actions_addopen (&factions, 1, outfile, flags,
94              S_IRWXU | S_IRWXG | S_IRWXO))
95              printf ("addopen out failed\n");
96      }
97
98      // Launch process!
99      int ret = posix_spawn (pid, name, &factions, &attr, pargs, environ);
100
101     // Delete attr
102     if (posix_spawnattr_destroy (&attr))
103         printf ("attr destroy failed\n");
104
105     // Delete file actions
106     if (posix_spawn_file_actions_destroy (&factions))
107         printf ("factions destroy failed\n");
108
109     // Done
110     return ret;
111 }
112
113 void Delay (unsigned int time)
114 {
115     struct timespec req;
116
117     req.tv_sec = time / 1000;
118     req.tv_nsec = (time % 1000) * 1000000;
119
120     nanosleep (&req, 0);
121 }
122
123 int Manager (int argc, char **argv)
```

```
124 {
125     // First, create the pipes
126     int man2proc, man2log, data;
127 
128     if ((man2proc = mkfifo ("/pipe/man2proc", 0x1A4)) < 0)
129         printf ("Error creating fifo man2proc\n");
130 
131     if ((man2log = mkfifo ("/pipe/man2log", 0x1A4)) < 0)
132         printf ("Error creating fifo man2log\n");
133 
134     if ((data = mkfifo ("/pipe/data", 0x1A4)) < 0)
135         printf ("Error creating fifo man2data\n");
136 
137     // Start the processing and logger programs
138     pid_t processingPid;
139     pid_t loggerPid;
140 
141     char *processingArgs[] = {"masters", "processor", 0};
142     char *loggerArgs[] = {"masters", "logger", 0};
143 
144     int a = CommandStartProcess (&processingPid, "/bin/masters", "/pipe/man2proc",
145         "/pipe/data", 0, processingArgs);
146     int b = CommandStartProcess (&loggerPid, "/bin/masters", "/pipe/data",
147         0, 0, loggerArgs);
148 
149     if (a < 0)
150     {
151         printf ("Failed to start processing\n");
152         return 1;
153     }
154 
155     if (b < 0)
156     {
157         printf ("Failed to start logger\n");
158         return 2;
159     }
160 
161     // Monitor the programs while checking for commands
162     while (1)
163     {
164         // Check for keyboard command
165         char c;
166         if (read (0, &c, 1) == 1)
167         {
168             if (c == 'q')
169             {
170                 printf ("Quitting\n");
171                 return 0;
172             }
173             else if (c == 'l')
174             {
175                 printf ("Sending message to logger\n");
176                 write (man2log, &c, 1);
177             }
```

```
178             else if (c == 'p')
179             {
180                 printf ("Sending message to processing\n");
181                 write (man2proc, &c, 1);
182             }
183         }
184 
185         // Check for status change
186         int status;
187         int pid = waitpid (-1, &status, WUNTRACED | WNOHANG);
188 
189         if (pid > 0)
190         {
191             if (pid == processingPid)
192             {
193                 if (WIFSTOPPED(status))
194                     printf ("The processing program has stopped\n");
195                 else
196                     printf ("The processing program has terminated\n");
197 
198                 for (int i = 0; i < 6; i++)
199                 {
200                     printf ("Restarting processing in %i...\n", 6 - i);
201                     Delay (1000);
202                 }
203 
204                 a = CommandStartProcess (&processingPid, "/bin/masters",
205                     "/pipe/man2proc", "/pipe/data", 0, processingArgs);
206 
207                 if (a < 0)
208                     printf ("Error: Failed to restart\n");
209             }
210             else if (pid == loggerPid)
211             {
212                 if (WIFSTOPPED(status))
213                     printf ("The logger program has stopped\n");
214                 else
215                     printf ("The logger program has terminated\n");
216 
217                 for (int i = 0; i < 6; i++)
218                 {
219                     printf ("Restarting logger in %i...\n", 6 - i);
220                     Delay (1000);
221                 }
222 
223                 a = CommandStartProcess (&loggerPid, "/bin/masters",
224                     "/pipe/data", 0, 0, loggerArgs);
225 
226                 if (a < 0)
227                     printf ("Error: Failed to restart\n");
228             }
229         }
230 
231         // Sleep some
```

```
232         Delay (10);
233     }
234 
235     // Done
236     return 0;
237 }
238 
239 int Logger (int argc, char **argv)
240 {
241     printf ("Starting the logger\n");
242 
243     int man2log = open ("/pipe/man2log", O_RDWR);
244 
245     if (man2log < 0)
246     {
247         printf ("Failed to open pipe\n");
248         return 1;
249     }
250 
251     char line[64];
252     int iter = 0;
253     int lastiter = 0;
254     char c;
255 
256     while (1)
257     {
258         // Check for command
259         if (read (man2log, &c, 1) == 1 && c == 'l')
260         {
261             printf ("Logger got signal to crash\n");
262 
263             volatile int *a = 0;
264             a = (volatile int*) 123;
265 
266             if (*a == 42)
267                 printf ("Logger failed to crash\n");
268         }
269 
270         // Check for work
271         int r = read (0, line, sizeof (line) - 1);
272 
273         if (r)
274         {
275             line[r] = 0;
276             printf ("Logger got msg at %i: %s", iter, line);
277             lastiter = iter;
278         }
279         else if (iter - lastiter > 3)
280             printf ("Logger warning: no work after %i iterations\n", iter - lastiter);
281 
282         // Wait for work
283         Delay (500);
284         iter++;
285     }
```

```
286
287     return 0;
288 }
289
290 int Processor (int argc, char **argv)
291 {
292     int state = 0;
293     char c;
294
295     printf ("Processing starting, state: %i\n", state);
296
297     while (1)
298     {
299         if (read (0, &c, 1) == 1 && c == 'p')
300         {
301             printf ("Processing got signal to crash\n");
302
303             volatile int *a = 0;
304             a = (volatile int*) 123;
305
306             if (*a == 42)
307                 printf ("Processing failed to crash\n");
308         }
309
310         state++;
311         printf ("Processing state: %i\n", state);
312         Delay (1000);
313     }
314
315     return 0;
316 }
317
318 ////////////////////////////////////////////////////////////////////////////////
319 // Usage
320 ////////////////////////////////////////////////////////////////////////////////
321 int PrintUsage (void)
322 {
323     printf ("Usage: masters [options]\n"
324            "  manager\n"
325            "    Start the test manager\n"
326            "  logger\n"
327            "    Start logger only\n"
328            "  processor\n"
329            "    Start processor only\n"
330            "  --help\n"
331            "    Print this message\n");
332     return 0;
333 }
334
335 int PrintUnknownCommand (void)
336 {
337     printf (
338         "Invalid argument, use\n"
339         "   masters --help\n"
```

```
340         "for a usage list\n");
341     return 1;
342 }
343
344
345 ////////////////////////////////////////////////////////////////////////////////
346 // Main function
347 ////////////////////////////////////////////////////////////////////////////////
348 int main (int argc, char **argv)
349 {
350     if (argc < 2)
351         return PrintUnknownCommand ();
352
353     if (!strcmp (argv[1], "manager"))
354         return Manager (argc, argv);
355     else if (!strcmp (argv[1], "logger"))
356         return Logger (argc, argv);
357     else if (!strcmp (argv[1], "processor"))
358         return Processor (argc, argv);
359     else if (!strcmp(argv[1], "--help"))
360         return PrintUsage ();
361     else
362         return PrintUnknownCommand ();
363 }
```

## B.5. kernel/kernel/src/mem/mem.c

```
 1 /*
 2  * Copyright (C) 2018 Erlend Sveen
 3  *
 4  * This program is free software: you can redistribute it and/or modify
 5  * it under the terms of the GNU General Public License as published by
 6  * the Free Software Foundation, either version 3 of the License, or
 7  * (at your option) any later version.
 8  *
 9  * This program is distributed in the hope that it will be useful,
10  * but WITHOUT ANY WARRANTY; without even the implied warranty of
11  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12  * GNU General Public License for more details.
13  *
14  * You should have received a copy of the GNU General Public License
15  * along with this program.  If not, see <https://www.gnu.org/licenses/>.
16  */
17
18 ////////////////////////////////////////////////////////////////////////////////
19 // Includes
20 ////////////////////////////////////////////////////////////////////////////////
21 #include "mem/mem.h"
22
23 ////////////////////////////////////////////////////////////////////////////////
24 // Variables
25 ////////////////////////////////////////////////////////////////////////////////
26 struct MemBank *membanklist = 0;
```

```
27
28 ////////////////////////////////////////////////////////////////////////////////
29 // Helper functions
30 ////////////////////////////////////////////////////////////////////////////////
31 struct MemBank *MemGetBank (unsigned int bankno)
32 {
33     struct MemBank *membank = membanklist;
34
35     for (int i = 0; i <= bankno; i++)
36         if (i == bankno)
37             return membank;
38         else if (membank->nextbank)
39             membank = membank->nextbank;
40
41     return membank;
42 }
43
44 struct MemBank *MemFindBank (unsigned int address)
45 {
46     struct MemBank *membank = membanklist;
47
48     do
49     {
50         if (address >= membank->bankaddress && address < membank->lastaddr)
51             return membank;
52
53         membank = membank->nextbank;
54     } while (membank);
55
56     return 0;
57 }
58
59 unsigned int MemSplit (struct MemBank *b, struct MemTableEntry *current,
60     unsigned int nbytes, int pos)
61 {
62     // Find an unused entry to split into
63     int foundi = 0;
64
65     for (int i = 0; i < MEM_LISTSIZE; i++)
66     {
67         if (!b->memlist[i].addr)
68         {
69             foundi = i;
70             break;
71         }
72     }
73
74     // Check
75     if (!foundi)
76         return 0;
77
78     // Get the entry
79     struct MemTableEntry *found = &b->memlist[foundi];
80
```

```
81     // Calculate the middle address
82     unsigned int middle = current->addr + nbytes;
83 
84     // Tricky part: New entry has to point to our current next
85     found->nextaddr = current->nextaddr;
86     found->addr = middle;
87     b->memilist[foundi] = b->memilist[pos];
88 
89     // Point ourselves to the new entry
90     current->nextaddr = middle;
91     b->memilist[pos] = foundi;
92     return 1;
93 }
94 
95 unsigned int MemAllocEntry (struct MemBank *b, struct MemTableEntry *current,
96     unsigned int blocksize, unsigned int nbytes, int pos)
97 {
98     // Copy
99     unsigned int addr = current->addr;
100 
101     // Check if we can split the block
102     if (blocksize >= nbytes)
103         if (!MemSplit (b, current, nbytes, pos))
104             return 0;
105 
106     // Set bit to indicate that the block is used
107     current->addr |= MEM_INUSEBIT;
108 
109     // Update statistics and return
110     b->currentfree -= nbytes;
111 
112     if (b->currentfree < b->lowestfree)
113         b->lowestfree = b->currentfree;
114 
115     return addr;
116 }
117 
118 unsigned int MemAllocMpuFindMinimal (unsigned int nbytes)
119 {
120     // Increment size by power of two
121     for (int i = 0; i < 32; i++)
122     {
123         // Calculate actual size
124         int regionsize = 1 << i;
125 
126         // Return if it is large enough
127         if (regionsize >= nbytes)
128             return i;
129     }
130 
131     // Error
132     return 0;
133 }
134 
```

```
135 unsigned int MemAllocMpuCalcRASR (unsigned int dataaddr,
136     unsigned int regionaddr,
137     unsigned int subregsize,
138     unsigned int nspare)
139 {
140     // Calculate subregion disable bits
141     unsigned int start = (dataaddr - regionaddr) / subregsize;
142     unsigned int nreg = 8 - nspare;
143     unsigned int reg = 0;
144
145     for (int i = 0; i < nreg; i++)
146         reg |= 0x0100 << (start + i);
147
148     return ~reg & 0x0000FF00;
149 }
150
151 void MemMerge (struct MemBank *b, unsigned char from, unsigned char to)
152 {
153     // Get current
154     struct MemTableEntry *current = &b->memlist[from];
155
156     // Get target entry
157     struct MemTableEntry *foundentry = &b->memlist[to];
158
159     // Check that they are in use
160     if (!(foundentry->addr & MEM_INUSEBIT) && !(current->addr & MEM_INUSEBIT))
161     {
162         // 'Consume' the next entry, set our nextaddr and ilist to its, and invalidate it
163         current->nextaddr = foundentry->nextaddr;
164         b->memilist[from] = b->memilist[to];
165
166         foundentry->addr = 0;
167         foundentry->nextaddr = 0;
168         b->memilist[to] = 255;
169     }
170 }
171
172 ////////////////////////////////////////////////////////////////////////////////
173 // Functions
174 ////////////////////////////////////////////////////////////////////////////////
175 void MemBankInit (struct MemBank *b, unsigned int address, unsigned int size)
176 {
177     b->bankaddress = address;
178     b->banksize = size;
179
180     for (int i = 0; i < MEM_LISTSIZE; i++)
181     {
182         b->memlist[i].addr = 0;
183         b->memlist[i].nextaddr = 0;
184         b->memilist[i] = 255;
185     }
186
187     b->currentfree = size;
188     b->lowestfree = size;
```

```
189     b->lastaddr = address + size;
190
191     b->memlist[0].addr = address;
192     b->memlist[0].nextaddr = b->lastaddr;
193
194     b->nextbank = 0;
195 }
196
197 void MemAddBank (struct MemBank *bank)
198 {
199     if (!membanklist)
200         membanklist = bank;
201     else
202     {
203         struct MemBank *membank = membanklist;
204
205         do
206         {
207             if (!membank->nextbank)
208             {
209                 membank->nextbank = bank;
210                 return;
211             }
212
213             membank = membank->nextbank;
214         } while (membank);
215     }
216 }
217
218 unsigned int MemAlloc (unsigned int nbytes, unsigned int bankno)
219 {
220     // Check for invalid argument
221     if (!nbytes)
222         return 0;
223
224     // Get bank
225     struct MemBank *b = MemGetBank (bankno);
226
227     if (!b)
228         return 0;
229
230     // Align the size
231     nbytes = (nbytes + 3) & ~3;
232
233     // Search list for free entries
234     // TODO: Use for loop
235     int pos = 0;
236
237     do
238     {
239         // Get the current memory block
240         struct MemTableEntry *current = &b->memlist[pos];
241
242         // Check if it is used and if it is the last block
```

```
243         if (current->addr & MEM_INUSEBIT)
244         {
245             if (current->nextaddr == b->lastaddr)
246                 return 0;
247 
248             pos = b->memilist[pos];
249             continue;
250         }
251 
252         // Get the size of the block and allocate if large enough
253         unsigned int blocksize = (current->nextaddr & ~MEM_INUSEBIT) - current->addr;
254 
255         if (blocksize >= nbytes)
256             return MemAllocEntry (b, current, blocksize, nbytes, pos);
257 
258         // Next
259         pos = b->memilist[pos];
260     } while (pos != 255);
261 
262     // No good block found
263     return 0;
264 }
265 
266 unsigned int MemAllocMpu (unsigned int nbytes, unsigned int bankno, struct MemMpuReturn *mpu)
267 {
268     // Check for invalid argument
269     if (!nbytes)
270         return 0;
271 
272     // Get bank
273     struct MemBank *b = MemGetBank (bankno);
274 
275     if (!b)
276         return 0;
277 
278     // Align the size
279     nbytes = (nbytes + 3) & ~3;
280 
281     // Minimum size for both Cortex-M and Cortex-R is 32 bytes
282     if (nbytes < 32)
283         nbytes = 32;
284 
285     // Get the nearest power of two that will fit nbytes
286     // Worst case it is nbytes*2-1 bytes
287     unsigned int mpuminimal = MemAllocMpuFindMinimal (nbytes);
288     unsigned int regionsize = 1 << mpuminimal;
289 
290     // Minimal allocation size and its subregion size
291     unsigned int minalloc = regionsize;
292     unsigned int subreg = regionsize / 8;
293 
294     // We may disable unused subregions to save memory, but
295     // the allocation has to fill the entire last subregion
296     if (minalloc > 128)
```

```
297     {
298         // Round nbytes to nearest multiple of subregion size
299         unsigned int n = nbytes / subreg;
300         n = n * subreg;
301 
302         // If it was rounded down, add a subregions worth of memory
303         if (n < nbytes)
304             n += subreg;
305 
306         minalloc = n;
307     }
308 
309     // At this point, we know that
310     //  1. The memory block has to be aligned with regionsize
311     //  2. minalloc has been rounded to subregion border size
312 
313     // Number of free subregions
314     unsigned int nspare = (regionsize - minalloc) / subreg;
315 
316     // Calculate mask
317     unsigned int mask = ~(regionsize - 1);
318 
319     // Search list for free entries
320     // TODO: Use for loop
321     int pos = 0;
322 
323     do
324     {
325         // Get the current memory block
326         struct MemTableEntry *current = &b->memlist[pos];
327 
328         // Check if it is used and if it is the last block
329         if (current->addr & MEM_INUSEBIT)
330         {
331             if (current->nextaddr == b->lastaddr)
332                 return 0;
333 
334             pos = b->memilist[pos];
335             continue;
336         }
337 
338         // Get the size of the block and allocate if large enough
339         unsigned int blocksize = (current->nextaddr & ~MEM_INUSEBIT) - current->addr;
340 
341         // Copy address
342         unsigned int addr = current->addr;
343 
344         // Calculate region start address
345         unsigned int regionaddr = addr & mask;
346 
347 redo:
348         // Check if it will fit
349         if (blocksize >= minalloc)
350         {
```

```
351            // Add up to nspare subregions until we are past the block start
352            unsigned int addrc = regionaddr;
353
354            for (int i = 0; i < nspare; i++)
355                if (addrc < addr)
356                    addrc += subreg;
357
358            // If addrc is still less than addr, we cannot use this block since
359            // it does not align with the MPU start address
360            if (addrc < addr)
361            {
362                // Go to the next region block, and add the gap to the allocation size
363                regionaddr += regionsize;
364                minalloc += regionaddr - addr;
365                goto redo;
366            }
367
368            // Add wasted subregions to the allocation amount and check
369            // that the block is still large enough
370            minalloc += addrc - regionaddr;
371
372            if (blocksize < minalloc)
373                goto redo;
374
375            // Set output
376            mpu->dataptr = addrc;
377            mpu->mpu_rbar = regionaddr;
378            mpu->mpu_rasr = MemAllocMpuCalcRASR (addrc, regionaddr, subreg, nspare);
379            mpu->mpu_rasr |= ((mpuminimal - 1) << 1);
380
381            // Check for split
382            if (blocksize >= minalloc)
383                if (!MemSplit (b, current, minalloc, pos))
384                    return 0;
385
386            // Set bit in structure
387            current->addr |= MEM_INUSEBIT;
388
389            // Update stats
390            b->currentfree -= nbytes;
391
392            if (b->currentfree < b->lowestfree)
393                b->lowestfree = b->currentfree;
394
395            // Clear memory and return
396            for (unsigned int i = addr; i < (addr + minalloc); i += 4)
397                *((unsigned int*)i) = 0;
398
399            return addr;
400        }
401
402        // Next
403        pos = b->memilist[pos];
404    } while (pos != 255);
```

```
405
406     // No good block found
407     return 0;
408 }
409
410 int MemFree (unsigned int addr)
411 {
412     // Find the memory bank
413     struct MemBank *b = MemFindBank (addr);
414
415     if (!b)
416         return 1;
417
418     // Loop for the address
419     for (int i = 0; i < MEM_LISTSIZE; i++)
420     {
421         // Get current entry and address
422         struct MemTableEntry *current = &b->memlist[i];
423         unsigned int entryaddr = current->addr & ~MEM_INUSEBIT;
424
425         // Is it our memory block?
426         if (entryaddr == addr)
427         {
428             // Clear in-use bit and update stats
429             current->addr = entryaddr;
430             b->currentfree += current->nextaddr - entryaddr;
431
432             // Get the next index and check for merge
433             unsigned char nexti = b->memilist[i];
434
435             if (nexti != 255)
436                 MemMerge (b, i, nexti);
437
438             // Also check for backwards merge
439             for (int j = 0; j < MEM_LISTSIZE; j++)
440             {
441                 if (b->memilist[j] == i)
442                 {
443                     MemMerge (b, j, i);
444                     break;
445                 }
446             }
447
448             return 0;
449         }
450     }
451
452     return 0;
453 }
```

## B.6. Cortex-R Context Switch

```
/*
 * Copyright (C) 2018 Erlend Sveen
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <https://www.gnu.org/licenses/>.
 */

.syntax unified
.cpu cortex-r4
.arm

.section .text.SchedContext
.global SchedContext
.type SchedContext, %function
SchedContext:
    cps #31              @ Supervisor mode: Access to everything but with user stack
    stmdb sp, {r0-lr}    @ Stack all the user regs except PC and CPSR, R0-LR is now free to use
    sub r0, sp, #60      @ Cannot have writeback in the above, sub 15 words and store in R0
    cps #18              @ Go back to IRQ mode, now have access to old PC and CPSR
    mrs r1, spsr         @ Get the CPSR from the banked reg
    stmdb r0!, {r1,lr}   @ Stack CPSR and PC

    ldr r1, =0xFFFFFFF4  @ Clear irq by reading SYS->SSIVEC
    ldr r2, [r1]

    ldr r3, =currentthc  @ Load address of currentthc then currentthc itself
    ldr r2, [r3]
    str r0, [r2]         @ Store the final stack pointer in the struct

.global SchedContextNext
SchedContextNext:        @ This entry is only used for crash recovery
    bl SchedSelectNext   @ Call scheduler selection function

    ldr r3, =currentthc  @ r0 becomes the stacktop member of currentthc
    ldr r1, [r3]
    ldr r0, [r1]

    add r1, r1, #8       @ Point r1 to our data in ThreadControl
    ldm r1, {r4-r9}

    cmp r4, #0           @ If the next task does not use the MPU, avoid setting the regs
```

```
52     beq skipmpu
53
54     mov r3, #4
55     mcr p15, 0, r3, c6, c2, 0     @ Set region
56     mcr p15, 0, r4, c6, c1, 0     @ Set address
57     mcr p15, 0, r6, c6, c1, 4     @ Set access rights
58     mcr p15, 0, r5, c6, c1, 2     @ Set size and enable
59
60     mov r3, #5
61     mcr p15, 0, r3, c6, c2, 0     @ Set region
62     mcr p15, 0, r7, c6, c1, 0     @ Set address
63     mcr p15, 0, r9, c6, c1, 4     @ Set access rights
64     mcr p15, 0, r8, c6, c1, 2     @ Set size and enable
65
66     add r1, r1, #24
67     ldm r1, {r4-r9}
68
69     mov r3, #6
70     mcr p15, 0, r3, c6, c2, 0     @ Set region
71     mcr p15, 0, r4, c6, c1, 0     @ Set address
72     mcr p15, 0, r6, c6, c1, 4     @ Set access rights
73     mcr p15, 0, r5, c6, c1, 2     @ Set size and enable
74
75     mov r3, #7
76     mcr p15, 0, r3, c6, c2, 0     @ Set region
77     mcr p15, 0, r7, c6, c1, 0     @ Set address
78     mcr p15, 0, r9, c6, c1, 4     @ Set access rights
79     mcr p15, 0, r8, c6, c1, 2     @ Set size and enable
80
81     @ These are the two extra regions
82     add r1, r1, #24
83     ldm r1, {r4-r9}
84
85     mov r3, #8
86     mcr p15, 0, r3, c6, c2, 0     @ Set region
87     mcr p15, 0, r4, c6, c1, 0     @ Set address
88     mcr p15, 0, r6, c6, c1, 4     @ Set access rights
89     mcr p15, 0, r5, c6, c1, 2     @ Set size and enable
90
91     mov r3, #9
92     mcr p15, 0, r3, c6, c2, 0     @ Set region
93     mcr p15, 0, r7, c6, c1, 0     @ Set address
94     mcr p15, 0, r9, c6, c1, 4     @ Set access rights
95     mcr p15, 0, r8, c6, c1, 2     @ Set size and enable
96
97 skipmpu:
98
99     ldmia r0!, {r1,lr}   @ Unstack CPSR and PC
100    msr spsr_csxf, r1    @ Put CPSR back in IRQ SPSR. csxf restores all the flags, not just cf.
```

```
101
102     @ See http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489f/CIHFIDAJ.html
103
104     cps #31             @ Switch to system mode
105     ldmia r0, {r0-lr}   @ Unstack R0-LR
106     cps #18             @ Switch back to irq mode
107     dsb                 @ Does this fix that memory access fault?
108     subs pc, lr, #4     @ Return from IRQ
109 .size SchedContext, .-SchedContext
```