



# Advanced Operating Systems

## Spring Semester 2022

Timothy Roscoe and the ETH Barelfish team

April 6, 2022

Hardware platform: Toradex Colibri i.MX8 board  
Architecture: ARMv8-A

# Preface

This is an advanced course about the design and implementation of operating systems. We've taught this course in one form or another at ETH Zurich for the last 10 years, inspired by (and originally based on) the Advanced Operating Systems course taught at the University of New South Wales by Kevin Elphinstone and Gernt Heiser. The course has a number of goals that we felt were not otherwise addressed by the Computer Science curriculum at ETH Zurich (and most other places).

Firstly, we want students to get experience with some of the key issues in OS design by actually building things.

A typical undergraduate course (and, indeed, textbook) on operating systems is quite theoretical. The reality of writing low-level software for a complex computer is quite different; some of the challenges are mentioned in textbooks (concurrency, asynchrony, etc.) but take on a whole new meaning when one actually has to implement the code. Others are rarely mentioned at all: the complexity of modern hardware, the absence of most high-level debugging tools, the need to implement sophisticated functionality without the benefit of a complex runtime, etc.

By actually designing bits of a different OS, and writing OS code, you get to solve common OS problems for yourself: how to deal with concurrency and asynchrony, tailor code to specific hardware, and how to architect a complete system of processes which work together to support applications.

Secondly, we want to give a perspective on OS design that's broader than Unix.

There is something of a dearth of good, up-to-date textbooks on operating system design, and the few good ones (e.g [6]) focus out of necessity on conventional, Unix-like designs. Few, if any, deal with non-Unix-like operating systems.

There are, of course, plenty of online tutorials on how to write your own operating system, but here the same thing often happens: there's a tremendous variety of ways of building parts of an OS, and yet almost all the online tutorials focus on how to build something that looks like Unix.

This is a pity, partly because this is a very impoverished view of what an OS design can be, partly because there are already some pretty good open-source versions of

Unix for PC-like hardware (Linux and FreeBSD, for example), and partly because if Unix is the only OS you know, it's hard to recognize where Unix might, in the context of modern hardware, get it wrong.

This, therefore, is not a course about Unix, Linux, or any Unix-like operating system. We avoid using a Unix-like OS to give a broader outlook on what an OS can be, and to foster a critical approach to OS designs. Hopefully, when you're done, you'll have a better understanding not just of the OS we're working with in this course, but also of Unix (or Linux, or Windows, or Android, or iOS, etc.).

Thirdly, we want to supply the historical background of many ideas in operating systems design, and convey a sense of what operating systems research is. The key ideas in OS design almost all came from research labs in industry and academia, and this perspective is missing from most online tutorials, and indeed many textbooks (Tanenbaum [97] is a notable exception). Also, we love doing research in OS design, and we'd like to share that sense of fun.

The structure of this book, therefore, weaves together at least four different types of material, which is usually labeled by icons in the margin.



Background

### Background

Historical origins and detailed discussions of the key OS concepts we talk about (capabilities, dispatch, paging, etc.). We won't reference standard textbooks much, but instead focus on research papers that talk about the ideas in detail. This material should give you more of a sense of what the ideas are, where they led, and what the design choices available might be should you end up encountering or using them in your project.



Technical Details

### Technical Details

Sometimes, we'll need to give fairly detailed descriptions of the way that some of the hardware or software works, or else give you help in navigating the other documentation for the platform. This type of section is going to be crucial for getting your project code to work – and figuring out why it doesn't work...



Project Instructions

### Project Instructions

These are the actual directions for each milestone: what you have to do, how you should go about it, and how to integrate what you've done into the rest of the system. This is the kind of material that in earlier versions of the course appeared in the form of milestone handouts, together with a bit more information about how we go about grading your work and testing your code.



## Commentary

This is stuff about the course itself: why is a particular section here? What are we trying to achieve with the course? How is it going to be structured? And, perhaps most importantly, bits of advice and hints and tips for going about the project work.



## Extra Challenge

### Extra challenges

Sometimes, we'll suggest work that isn't necessary for the course, but might be interesting and has the opportunity to win you extra points. We strongly suggest you get the rest of each milestone working before attempting any extra challenges, but if you do decide to go for these, and succeed, make sure you point them out to us in the grading session.

### Code

One more lexical convention in this book is presenting code and console output. We will write C code fragments as follows:

```

1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     printf("hello, world\n");
6     return 0;
7 }
```

Interaction with the command line for building and booting the OS is shown like this:

```
$ echo 'hello, world'
hello, world
$
```

Finally, raw console output from your OS will be displayed as follows:

```

1 [ aboot second-stage loader ]
2
3 Entry point is 8000E980
4 Reading 278756 bytes to 80000000
5 ...
6 ...
7
8 kernel ARMv7-A: Welcome to AOS.
```

## Acknowledgments

Finally, a word of thanks. Many people deserve thanks for helping to develop this course. A number of ETH Zurich Masters students, teaching assistants, PhD students, postdocs, and visitors have helped over the years to develop this course, and we are extremely grateful to all of them.

A few in particular deserve special mention. Kevin Elphinstone and Gernot Heiser at the University of New South Wales developed the course on which this was originally based in 2008, Andrew Baumann was instrumental in bringing it to ETH Zurich and co-teaching it, and Simon Peter was the first teaching assistant for it.

Tom Anderson, Stefan Kaestle, Pravin Shinde, and Akhilesh Singhania provided much early input and feedback.

Simon Gerber, Gerd Zellweger, Reto Achermann, Lukas Humbel, David Cock, and Moritz Hoffmann took on the difficult job of revamping the course and transferring it from L4 to Barrelyfish.

Since then, Daniel Schwyn, Roni Häcki, and Nora Hossle have helped greatly with keeping the course up to date and fixing bugs.

Finally, students taking the course (including you, hopefully) have helped a great deal over the years with feedback, bug reports, and helpful suggestions.

# Contents

<b>Preface</b>	i
<b>1 Introduction</b>	1
1.1 Operating Systems research today . . . . .	1
1.2 Barrels . . . . .	4
1.3 What do we assume? . . . . .	5
1.4 Structure of the course . . . . .	7
1.5 The Toradex board . . . . .	9
1.6 The ARMv8-A Architecture . . . . .	14
1.7 Grading Scheme . . . . .	16
1.8 Previous grade distributions . . . . .	17
1.9 Hints and Tips . . . . .	18
<b>I Individual early milestones</b>	21
<b>2 Getting started</b>	23
2.1 Check you've got the hardware . . . . .	23
2.2 Check you've got the toolchain . . . . .	25
2.3 Check out the project code . . . . .	27
2.4 Creating a build directory and Makefile . . . . .	27
2.5 What happened? What's this Hake thing? . . . . .	28
2.6 Test the boot tool . . . . .	29
2.7 Understanding Booting . . . . .	32
2.8 Test the Virtual Machine . . . . .	34
2.9 What just happened? . . . . .	35
2.10 Building your own kernel . . . . .	35
2.11 Console output . . . . .	38
2.12 A Simple Interpreter/Shell . . . . .	39
2.13 Flash the LED . . . . .	39
2.14 System Shutdown/Reset . . . . .	40
2.15 Milestone 0 Summary . . . . .	41
<b>3 Memory management and capabilities</b>	43

3.1	The first real milestone . . . . .	43
3.2	Memory management in general . . . . .	43
3.3	What are capabilities? . . . . .	45
3.4	Implementing capabilities . . . . .	50
3.5	Capabilities in Barreelfish . . . . .	54
3.6	Getting prepared . . . . .	57
3.7	Bugs! . . . . .	58
3.8	Error handling in Barreelfish . . . . .	59
3.9	The Barreelfish capability API . . . . .	60
3.10	Task 1: a physical frame allocator . . . . .	62
3.11	Support for partial frees . . . . .	64
3.12	The ARMv8-A MMU . . . . .	64
3.13	Constructing safe page tables . . . . .	68
3.14	Task 2: mapping frame capabilities . . . . .	70
3.15	Superpage Mappings . . . . .	72
3.16	How we test . . . . .	72
3.17	How to do well . . . . .	73
3.18	Milestone 1 Summary . . . . .	73
<b>II</b>	<b>Group work: Building the core of the OS</b>	<b>75</b>
<b>4</b>	<b>Processes, threads, and dispatch</b>	<b>79</b>
4.1	Process models . . . . .	79
4.2	Kernel threading models . . . . .	80
4.3	Barreelfish kernel architecture . . . . .	85
4.4	Creating a process . . . . .	85
4.5	Fork or spawn? . . . . .	86
4.6	Extend the paging code . . . . .	87
4.7	Unmapping . . . . .	88
4.8	Find and map the ELF binary . . . . .	88
4.9	Setup initial C- and V-Space . . . . .	90
4.10	Passing Address Space Information . . . . .	92
4.11	Parsing the ELF . . . . .	92
4.12	User Thread Models . . . . .	93
4.13	A moment of Zen: threads and stacks . . . . .	101
4.14	Dispatch in Barreelfish . . . . .	102
4.15	Challenging hack: resuming a thread . . . . .	104
4.16	Set up the dispatcher . . . . .	108
4.17	Set up arguments . . . . .	109
4.18	Start the process at last . . . . .	110
4.19	Note on Hakefiles . . . . .	110
4.20	Process Management . . . . .	111
4.21	How to do well . . . . .	113
4.22	Milestone 2 Summary . . . . .	113

<b>5 Message passing</b>	<b>115</b>
5.1 The Basics . . . . .	115
5.2 Local Remote Procedure Call . . . . .	117
5.3 Lightweight Remote Procedure Call . . . . .	119
5.4 IPC in L4 . . . . .	121
5.5 Lightweight message passing in Barrelyfish . . . . .	124
5.6 LMP programming interface . . . . .	125
5.7 Stack ripping . . . . .	127
5.8 The milestone: overview . . . . .	128
5.9 Getting prepared . . . . .	129
5.10 Implementing your channels . . . . .	129
5.11 Passing large messages . . . . .	134
5.12 A memory server . . . . .	135
5.13 Spawning Domains . . . . .	136
5.14 A terminal driver . . . . .	136
5.15 Optimization . . . . .	136
5.16 How to do well . . . . .	137
5.17 Milestone 3 Summary . . . . .	137
<b>6 Page fault handling</b>	<b>139</b>
6.1 A different view of virtual memory . . . . .	139
6.2 The classic Unix memory API . . . . .	140
6.3 Stepping back a bit . . . . .	142
6.4 User-managed virtual memory . . . . .	143
6.5 How does it all work? . . . . .	144
6.6 Implement an Exception Handler . . . . .	146
6.7 Design the Address Space Layout . . . . .	146
6.8 Programming the MMU . . . . .	148
6.9 Implement Page Fault Handling . . . . .	148
6.10 Dynamic stack allocation . . . . .	149
6.11 Dynamic capability slot and data structure allocation . . . . .	150
6.12 Background . . . . .	150
6.13 How we test . . . . .	152
6.14 How to do well . . . . .	152
6.15 Milestone 4 summary . . . . .	153
<b>7 Multicore</b>	<b>155</b>
7.1 A concrete example: the Toradex board . . . . .	155
7.2 Early multiprocessors . . . . .	156
7.3 Multics . . . . .	156
7.4 Firefly . . . . .	158
7.5 Multicore . . . . .	159
7.6 Scaling an OS to multiple cores . . . . .	160
7.7 Scalable locks . . . . .	161
7.8 Tornado and K42 . . . . .	163
7.9 Scalable commutativity . . . . .	166

7.10 Barreelfish . . . . .	167
7.11 Booting a secondary core in Barreelfish . . . . .	170
7.12 Getting prepared . . . . .	171
7.13 Bringing up a second core . . . . .	172
7.14 Booting All Cores . . . . .	175
7.15 Suspend & Resume . . . . .	175
7.16 Inter-Core Communication . . . . .	175
7.17 Multicore Memory Management . . . . .	177
7.18 Experience with Barreelfish . . . . .	178
7.19 How to do well . . . . .	179
7.20 Milestone 5 summary . . . . .	179
<b>8 User-level message passing</b>	<b>181</b>
8.1 Traditional communication . . . . .	181
8.2 Ring buffers and shared memory . . . . .	182
8.3 User-level RPC . . . . .	183
8.4 A quick recap on cache coherency . . . . .	186
8.5 UMP operation . . . . .	188
8.6 Message passing vs. shared memory . . . . .	191
8.7 Memory consistency . . . . .	193
8.8 Getting prepared . . . . .	195
8.9 Sharing a Frame Between Cores . . . . .	196
8.10 Establishing a communication protocol . . . . .	196
8.11 Fragmentation and reassembly . . . . .	197
8.12 Performance . . . . .	198
8.13 Binding . . . . .	198
8.14 Arbitrary channels . . . . .	199
8.15 How to do well . . . . .	200
8.16 Milestone 6 summary . . . . .	200
<b>III Individual projects: building on the core</b>	<b>201</b>
<b>9 A Shell</b>	<b>205</b>
9.1 Keyboard input . . . . .	206
9.2 Implement a command-line interface . . . . .	208
9.3 Process Management . . . . .	209
9.4 Implement I/O redirection and pipes . . . . .	210
9.5 Network login . . . . .	210
9.6 How to do well . . . . .	210
9.7 Summary . . . . .	210
<b>10 Filesystem</b>	<b>213</b>
10.1 Setting up the block driver . . . . .	213
10.2 Testing the block driver . . . . .	214
10.3 A faster block driver . . . . .	214

10.4 The filesystem itself . . . . .	215
10.5 Add the support for long file entries . . . . .	216
10.6 Design Space . . . . .	216
10.7 Implementing a file system . . . . .	216
10.8 Performance Evaluation . . . . .	218
10.9 VFS, MBFS, NFS, ... . . . . .	218
10.10 How to do well . . . . .	220
10.11 Summary . . . . .	220
<b>11 Networking</b>	<b>221</b>
11.1 Overview . . . . .	221
11.2 Network hardware . . . . .	222
11.3 Starting the driver . . . . .	224
11.4 Sending/Receiving raw packets . . . . .	224
11.5 Implementing ARP . . . . .	226
11.6 Reply to echo requests (ping) . . . . .	227
11.7 Debugging hints . . . . .	227
11.8 UDP echo server . . . . .	228
11.9 Multiple clients . . . . .	228
11.10 Performance measurement . . . . .	229
11.11 Remote UDP login . . . . .	229
11.12 TCP . . . . .	229
11.13 How to do well . . . . .	230
11.14 Summary . . . . .	230
<b>12 Nameserver</b>	<b>231</b>
12.1 Getting prepared . . . . .	232
12.2 Registration . . . . .	233
12.3 Rich properties . . . . .	234
12.4 Dead service removal . . . . .	234
12.5 Lookup . . . . .	234
12.6 Bootstrap . . . . .	236
12.7 Enumeration . . . . .	236
12.8 Getting the rest of the system to use the nameserver . . . . .	237
12.9 How to do well . . . . .	237
12.10 Interface to implement . . . . .	237
12.11 Shell Integration to Implement . . . . .	238
12.12 Summary . . . . .	238
<b>13 Capabilities revisited</b>	<b>239</b>
13.1 Background: Distributed capabilities . . . . .	240
13.2 Getting prepared . . . . .	240
13.3 Transferring capabilities between the init processes on both cores . . . . .	240
13.4 Ownership transfer . . . . .	242
13.5 Forward libaos capability operations that need synchronization to init on the same core . . . . .	242

13.6 Deleting CNodes and Dispatchers . . . . .	242
13.7 Revoking capabilities . . . . .	243
13.8 Deleting capabilities that exist on both cores . . . . .	244
13.9 Revoking capabilities that exist on both cores . . . . .	244
13.10 Retype capabilities that exist on both cores . . . . .	244
13.11 Use standard capability operations from init . . . . .	245
13.12 Use “real” capability transfer when initializing init on core 1 . . . . .	245
13.13 Send capabilities between any two domains . . . . .	245
13.14 How to do well . . . . .	245
13.15 Summary . . . . .	246
<b>IV Group work: The Report</b>	<b>247</b>
<b>14 The Report</b>	<b>249</b>
14.1 Structure of the report . . . . .	249
14.2 Group Milestones . . . . .	249
14.3 Documenting a design . . . . .	250
14.4 Documenting the process . . . . .	252
14.5 Individual project write-ups . . . . .	252
14.6 Performance evaluation . . . . .	253
14.7 Benchmarks . . . . .	253
14.8 Performance methodology . . . . .	254
14.9 Presenting performance results . . . . .	255
14.10 How fast does it need to be? . . . . .	255
14.11 How to do well . . . . .	256
14.12 Final milestone summary . . . . .	256
14.13 A final word . . . . .	257
<b>V Handing in your Code</b>	<b>259</b>
<b>15 Preparing Final Submission</b>	<b>261</b>
<b>Bibliography</b>	<b>261</b>

# Chapter 1

## Introduction

Welcome to Advanced Operating Systems!

This course is about the principles, design, implementation, and analysis of operating systems well beyond the ideas in the Linux (or other Unix-derived) kernel. It's deliberately different, and motivated by the realization that the world is changing rapidly for operating systems design.

### 1.1 Operating Systems research today

Trends in modern hardware make this an interesting time in OS design, although most of the research community has been somewhat slow to realize this. Figure 1.1 shows the kind of diagram of a modern computer that one frequently encounters in an OS textbook.

There are so many things wrong or misleading about this kind of diagram that it's hard to know where to start listing them. If you remove the graphics and USB hardware, this might plausibly resemble an (unexpanded) DEC VAX 11/780 from 1977, or perhaps an Intel 80386-based PC from 1985. Even then, it's missing a network interface (arguably a more critical peripheral component of the Xerox Alto in 1973 than the display).

There's no processor cache (which operating systems have to manage). There's only one processor. The "system bus" is, well, a bus – not a network as it is on any modern phone System-on-Chip (SoC), PC server, or mainframe. The disk controller is still around, but it's more likely today that an SSD is connected to it rather than a set of spinning ceramic disks. Increasingly, though, there's an NVMe controller instead directly attached to the I/O interconnect, and this affects OS design heavily because the bandwidth to and from NVRAM is getting to approach that of the main interconnect – indeed, the bandwidth of high-end network cards can exceed that of the system interconnect. The graphics adapter is, itself, a full-blown



*Background*

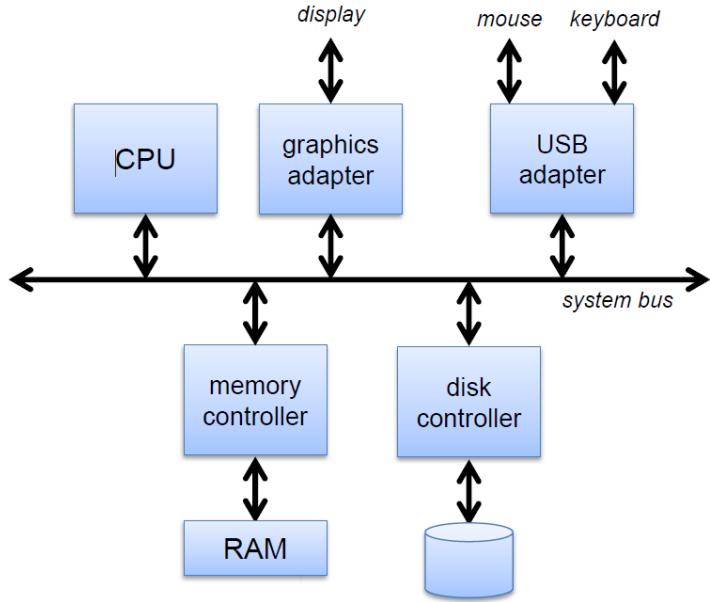


Figure 1.1: Not a real computer

highly parallel processor, which might not actually be used for the display at all (as in high-performance computing applications).

The main reason why a modern computer doesn't look at all like this diagram is the end of Moore's Law and related trends in hardware. For many years, mainstream computers had essentially the same architecture but got faster year-on-year because transistors become exponentially more plentiful. More, and smaller, transistors meant more complex processors and higher clock frequencies.

The first obstacle to this progress has been called the *power wall*. The power dissipated by a CMOS circuit is the sum of the *static power* (the leakage whenever the circuit is powered) and *dynamic power* (the energy dissipated when a transistor switches):

$$P_{total} = P_{dyn} + P_{static}$$

Roughly speaking, the dynamic power is proportional to the capacitance of the circuit, the square of the voltage, and the frequency (how often it actually switches):

$$P_{dyn} \approx C \times V^2 \times f$$

The static power is generally proportional to the supply voltage. This means that

with increases in clock frequency, more energy is dissipated. Eventually, it becomes hard to carry this energy away from the chip fast enough. Decreasing the supply voltage works to some extent, but as the voltage continues to drop, the rate at which a gate can be switched is reduced and eventually there is too little potential different to prevent static leakage across the transistor gate, and static power dissipation starts to dominate. We've basically hit the practical power limit for cooling commodity microprocessors, and that's the reason that clock frequencies have not been rising for the last 10 years or so, and in many cases have been falling.

The second obstacle is sometimes called the *ILP wall*. Instruction Level Parallelism is the ability to extract parallelism from a single threaded sequence of machine instructions, via aggressive reordering, register renaming, deep pipelining, super-scalar execution, etc. Historically, this was where a lot of the transistors made available by the progress of Moore's law were spent, creating immensely complex CPU cores (which also consumed a lot of power).

However, there's only a limited degree of parallelism theoretically present in a given single thread of execution, and so ILP eventually delivers diminishing returns with increased core complexity. For the most part, serial performance acceleration using ILP has stalled. Another, related, factor is the sheer complexity of the fastest, aggressively out-of-order cores – simply validating the correctness of the processor (nobody wants to ship CPUs with bugs) could take many months of continuous simulation running on hundreds of thousands of machines.

Both these obstacles have, in the short term, led to a halt in efforts to make processors faster and instead to efforts to put more processing cores on a single die – multicore. It is now quite hard to find a single-core processor for sale. This means that most software, and certainly any operating system, has to assume that it will be managing a growing number of processor cores.

The third obstacle is the *memory wall*: processors have been getting faster in terms of the number of instructions they can execute and the amount of data they can consume in unit time. Dynamic RAM has also been getting faster, in terms of the numbers of reads and writes it can sustain in unit time. The problem is that processors have been getting faster much faster than memory. At some point, the effective speed of the processor is limited by the speed of the memory system.

This has led to ever-more-complex memory systems: larger caches, deeper cache hierarchies (3 levels of cache is not unusual in modern microprocessors) dividing memory into banks on independent controllers so that processors can access different memory regions concurrently, or memory bandwidth is increased by striping or interleaving accesses across controllers and banks, and Non-Uniform Memory Access, where clusters of cores have locally attached DRAM which is faster for them to access than DRAM attached to other groups of cores in the system.

For an OS, the consequences are that caches must be extensively managed (indeed, many have proposed limiting cache coherence to "islands" in the machine to save power and increase performance), memory allocation has to be much better informed since DRAM now has many different types and memory locations, etc.

Unfortunately for Figure 1.1, this is only the beginning. Ultimately, putting more cores on a chip doesn't solve the basic problem that the transistors aren't getting any faster, and are generating too much heat. Until about 2006, the *power density* of transistors stayed pretty much constant, which meant (very roughly) that you could pack more smaller transistors into the same space and they would consume the same power ("Dennard scaling"). That doesn't seem to be true any more, and the implication is that most of a chip in the future will have to be powered down most of the time – referred to as "Dark Silicon" [46].

This has far-reaching implications for OS design that, at time of writing, are barely understood at all. What does seem to be clear is that getting more performance from dark silicon circuits involves using the transistors in increasingly specialized ways, so that the limited number of circuits that can be powered at any one time do their required job fast and efficiently before being switched off and others turned on [98].

It's also not just about cores. There are similar power-related limits on the size of Dynamic RAM (which has to be continually powered-on and refreshed), but which might not be shared by non-volatile memory (which can be turned off most of the time). It seems likely that computers in the future will have very large, non-volatile, byte-addressable main memories [18].

Unix was not remotely designed for this kind of hardware landscape. In its current form, it simply cannot handle heterogeneous cores, non-coherent shared memory, non-shared memory, or very large non-volatile main memories, let alone the rapid switching of functionality envisioned in dark silicon.

This makes it an interesting time in OS research and design, and a good time to revisit many key ideas in the evolution of OS designs which have either remained in research, or were widely deployed but fell into disuse with the rise of Unix and VAX-like architectures as the dominant model of computing.



Technical  
Details

## 1.2 Barreelfish

This course is based on an operating system called Barreelfish, which has been developed at ETH Zurich over the last 10 years or so, with support from a number of companies including ARM Ltd, Texas Instruments, Cisco Systems, Hewlett-Packard Enterprise, Huawei, Intel, and VMware. The OS was started as a joint project between ETH Zurich and Microsoft Research. It's open source (MIT license), and can be downloaded from <http://www.barreelfish.org/>.

Barreelfish was originally written to investigate the challenges posed by modern hardware: multiple cores, a wide range of different and complex systems and devices, heterogeneity in instruction sets, and scaling to large memories. Since then, it's become a useful vehicle for a variety of research and teaching projects at ETH and elsewhere. It's actively maintained and developed by the group at ETH Zurich, and has a number of external users. If you still interested in Barreelfish after taking

this course and would like to get more involved, please get in touch.

Today, Barreelfish supports 64-bit Intel and ARMv8-A architecture processors, plus 32-bit ARMv7-A and 64-bit Intel Xeon Phi cores. It remains a research OS, which means that it's not something that is likely to replace Windows, MacOS, iOS, Android, or Linux any time soon. However, Barreelfish is still capable of running a number of applications, such as databases, a shell, an SSH server, a web server, a variety of benchmark suites, and a virtual machine monitor capable of booting a Linux kernel inside a virtual machine. A version has even run Microsoft Office 2013, using the Drawbridge system [89].

We use Barreelfish in this course for several reasons. The first, and main, reason is that it is very different to OS designs derived (more or less) from Unix, such as Linux, BSD, MacOS, iOS, Android, and Windows. This “shock of the new” provides a different perspective on both OS research, design, and implementation in general, and also on Unix as one specific set of design choices.

Barreelfish is also a much smaller OS than Linux. In particular, as part of its design, it uses a small, compact kernel (the “CPU driver”) which is single-threaded, and non-preemptable – it can be thought of as a program which sequentially handles events (system calls, traps, interrupts, etc.) and runs each to completion in a short time. This makes it much easier to understand. Barreelfish also uses a different approach to supporting multiple cores: its key feature is that it runs a different CPU driver on every core, pushing any inter-core synchronization code to user space (the “multikernel” architecture).

Finally, Barreelfish uses a lot of ideas from older research OSes, and so is handy for demonstrating many ideas in practice which are absent or incompletely implemented in systems like Linux. We'll discuss some of these as the course progresses: capabilities for resource management (from KeyKOS [54] and seL4 [40]), fast interprocess communication (from LRPC [27] and L4 [76]), minimizing shared state (from Tornado [49] and K42 [67]), dispatching processors using upcalls (from Psyche [79] and Scheduler Activations [7]), pushing OS policy into user space (from Exokernel [44] and Nemesis [72]), user-space RPC decoupled from IPIs (from URPC [27]), and running drivers in their own processes (from microkernels [75, 83] and Xen [19]), and specifying device registers in a little language (from Devil [81]).

## 1.3 What do we assume?

You don't have to know anything already about Barreelfish, or ARM assembly language, to take this course. However, like any other course, this one does assume you have some background.

Firstly, programming: we'll mostly be working in **C**, with some assembly language. To succeed in the course, you have to be pretty comfortable with programming in C (as opposed to C++, C#, Java, Perl, Python, etc.). Good C programming always entails a keen sense of what is happening to memory and registers underneath,



Commentary

so by “pretty comfortable with programming in C”, we mean that you know what pointers are, how the address space is laid out, how storage is allocated on both the stack and on the heap, and how structures and stack frames are formatted in memory.

Secondly, we assume you know a fair bit about **how hardware works**, at least from the viewpoint of software: caches (both virtual and physical), Memory Management Units (MMUs) and page tables, devices and memory-mapped I/O, interrupts, direct memory access (DMA), etc. You don’t need to know anything in advance about the specifics of the platform we’ll be using – we supply detailed technical reference material – but you do need to be able to understand and navigate this documentation to figure out the specifics for yourself.

Thirdly, you need to know some **assembly language**. The exact processor architecture or architectures you know about doesn’t matter a lot – again we supply reference material, and the amount of assembly language you’ll need to write is pretty minimal – but you will need to understand listings, particularly when debugging.

At ETH Zurich, we teach a class to 2nd year (3rd semester) undergraduates called “Systems Programming and Computer Architecture”. If you’ve taken this course, or something like it (it is partly based on the famous CS 15213 at Carnegie Mellon), you should be fine.

Fourth, this is an *advanced* course on operating systems, which implies that we assume you already know a fair amount about **operating systems** at a basic level. This includes theoretical concepts like concurrency (multiple executions happening at the same time), asynchrony (things happening unexpectedly), parallelism (exploiting concurrency to increase performance), multiprogramming (creating the illusion of concurrency using context switching), and virtualization (creating the illusion of a different execution environment). It also includes more practical material like paging, scheduling, device drivers, process management, kernel vs. user space execution, etc.

At ETH Zurich, we teach parallel programming and concurrency in the first year of the undergraduate curriculum, and operating systems later on (after Systems Programming). These courses are our guideline for what we assume you know; any fairly practical OS course that covers the design and implementation of Unix, and any basic undergraduate concurrency material that includes programming with threads should be sufficient.

Finally, most of this course centers on developing code. The development environment we support is **Linux** (specifically, Ubuntu 20.04 LTS), and you should feel comfortable with the Linux shell, text editors, and tools like `make`, `gcc`, etc. In the past, students have managed to work on Windows or MacOS (using a combination of native tools, ports, and virtual machines), but for this you’ll be on your own.



## 1.4 Structure of the course

The course is structured in a sequence of project milestones, each one accompanied by background readings (see Figure 1.2).

The project milestones mostly build on one another, with the goal at the end of having a complete, albeit simple, operating systems that can run small applications. Some of the milestones are done individually (i.e., on your own), but the central part of the course where the core of the OS is built is intended to be done as a team of 3-4 people. Figure 1.2 shows how the various milestones of the course fit together.

Each week, we'll have two consultation sessions - the first is for discussing the background concepts and history for the course, while the second is in a lab and is a chance to talk about specific implementation issues you might encounter. This is also where you demonstrate the milestones, and we grade the progress you've made in the course.

The first milestone is described in the next chapter, and is about familiarizing yourself with the target hardware, the software tool chain, booting a test image on the platform, and some initial kernel programming.

The second milestone is about physical memory management, and introduces one of the key concepts in the course: capabilities as a way of managing, allocating, accounting, and authorizing access to physical memory and other resources.

Following this, you'll form teams and work on the central features of the OS through 5 milestones: process creation, inter-process communication using message passing, handling page faults, booting multiple cores, and message passing between processes on different cores. Each of this is done in a rather different way to Unix-like systems such as Linux, MacOS, or BSD.

After this, each member of the team will take up an individual project which builds on the basic OS you've built. You get some choice here, and the options include a file system, network stack, shell, name server, and others. Three or four of these fit together to provide a complete system you can demonstrate running your programs.

The final milestone is to write a report on the whole project - this is done by the whole team, but we expect the chapters on the last, individual projects to be written by whoever implemented them. This report is an important deliverable for the course, so don't treat this as an afterthought! It's a good idea to write it as you go along, for several reasons.

One is that we are looking for much more than documentation for the system, and also much more than a simple description of what you did. We want to know how you approached the problems, what alternatives you considered, why you chose the approaches you did and how you arrived at that decision. Good reports in the past have included performance graphs to justify particular design features, diagrams illustrating tradeoffs, discussions of ideas which seemed like a good idea at the time but eventually had to be abandoned and replaced, etc.

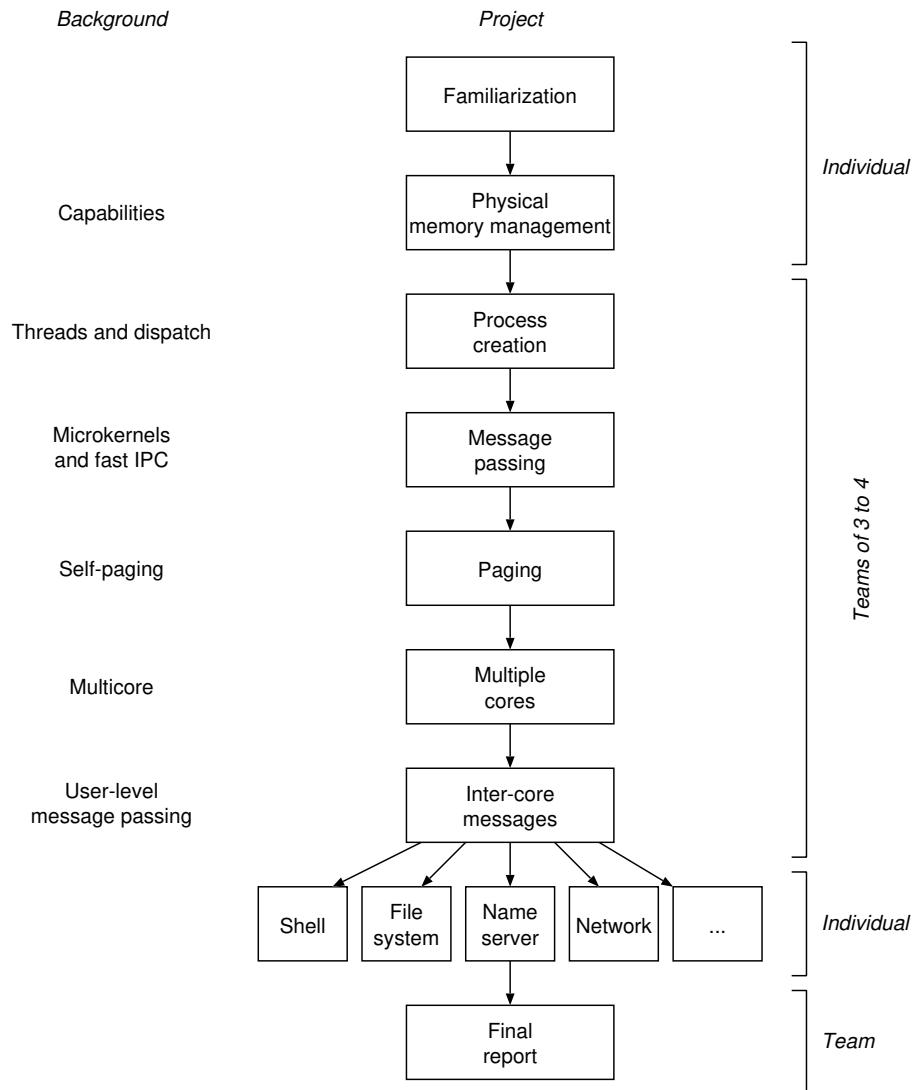


Figure 1.2: Course structure



Figure 1.3: The Toradex Colibri iMX8X module

The course as a whole is graded in several parts. There are marks for delivering and demonstrating each milestone, and late milestones incur a penalty, so beware milestones which build on previous ones and try not to slip behind. In addition, at the end of the course we will run a collection of automated unit tests against your OS, and some marks are derived from these test results. Finally, there are a significant number of marks for the quality, completeness, and depth of the final report at the end.

## 1.5 The Toradex board

You'll be working with real hardware in this course. The platform we'll use is the Toradex iMX8X module, shown in Figure 1.3.

As you can see, it's pretty small. It's an SODIMM form factor, with almost no other connectors. Boards like this are produced for embedded systems, where the module tends to plug into some other board for interfacing.

We'll use the Aster carrier board for this, which is also produced by Toradex (see Figure 1.4).

The combination doesn't come with a lot of packaging. As a result, please be careful when plugging or unplugging anything – it's easy to break the hardware if you're not gentle with it.

The Toradex board has a lot of connectors, but for the moment we only need to know about a few.

First, there are two micro-USB connectors. One of them (below the USB Type-A



Technical  
Details

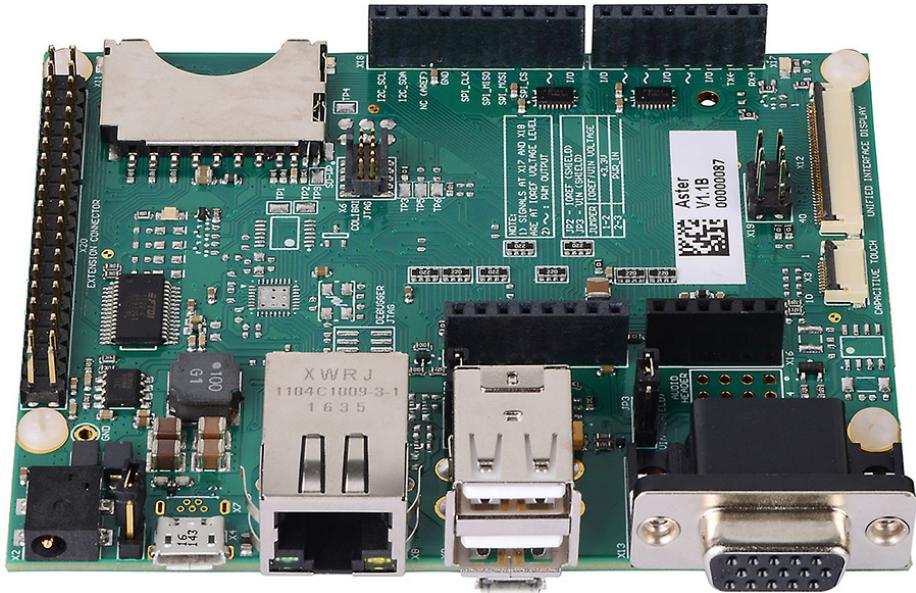


Figure 1.4: The Toradex Aster carrier board. The iMX8X plugs into the underside.

connectors) provides a “USB On-The-Go” port. This is how you boot the board: you’ll compile a boot image on your PC, and copy it over this port into system’s memory before jumping to it and booting your code. The other microUSB port powers the board (there’s a 5V DC power connector you can use, but we won’t in this project).

This makes things a lot simpler early on in the project: you don’t need power cables, network connectivity, an SD card, or much else plugged into your board, just two USB cables between it and your PC.

For the console for the OS, we will use a Universal Asynchronous Receiver/Transmitter (UART) interface. This is a very old, but much-loved way to send a stream of characters (up to 115 kbits/second) to and from a device over what used to be known as an RS-232C serial interface. Pretty much all computers still provide a UART, but increasingly these days you only see a physical connector for it (often a 9-way D connector) on server machines.

The NXP processor SoC actually has a number of UARTs, but the one we’ll use to start is conveniently connected to the same microUSB port that powers the board.

The UART is still, for most people bringing up an OS on new hardware, the first device you write a simple driver for. When you execute `printf()` in the kernel on the Toradex board, the results should come out of this port and get displayed in a console window on your PC.

Two other features of the board are worth pointing out right now. The first is the

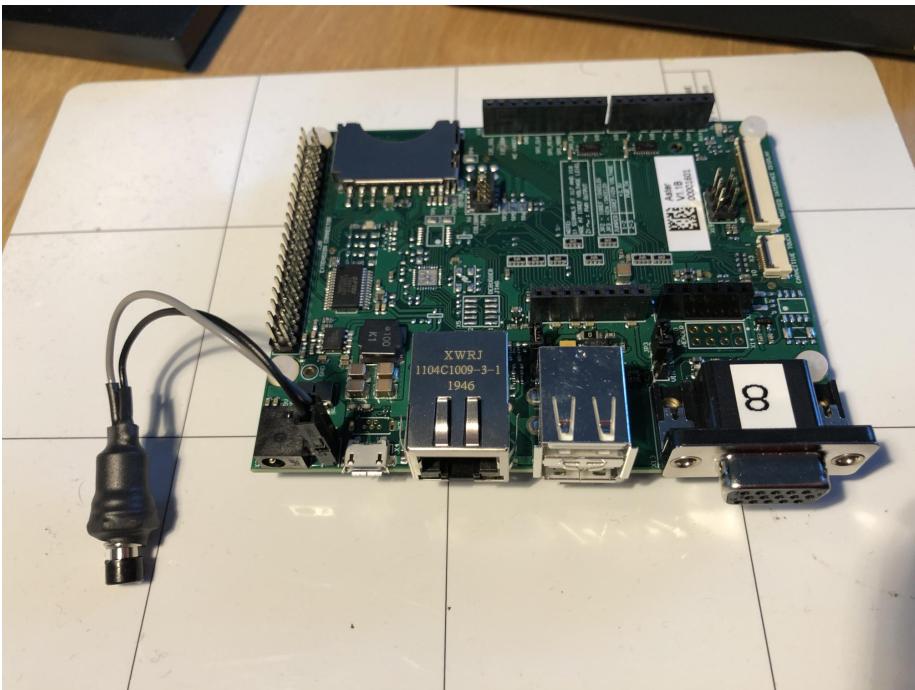


Figure 1.5: Aster board with the reset switch connected to the power jumper

reset button. This dangles off your board connected to a pin header with two jumper wires right next to the DC power connector in the corner (Figure 1.5). Pressing the button cuts power (by switching the power source from the USB port to the 5V DC connector and back) which is a convenient way of resetting a system without unplugging anything.

There are also some LEDs (see the data sheet [103] for where to find them on the board). One of these (LED1) is the power indicator. When you plug your board into something via the second microUSB port, this should light up (if not, there may be a problem with your board).

The other one is controllable by software. We'll be flashing this LED under software control later.

Figure 1.6 shows the high-level architecture of the Aster board, and is taken from the data sheet [103]. However, it's clear that most of the interesting stuff is on the main SoC, the i.MX 8X from NXP.

Understanding the manual [84] for this chip is important to getting things working during the course, but it can be pretty intimidating if you have never worked with this kind of documentation before. For one thing, it's more than 8,600 pages long, and that doesn't include any information on the processor cores themselves, or the

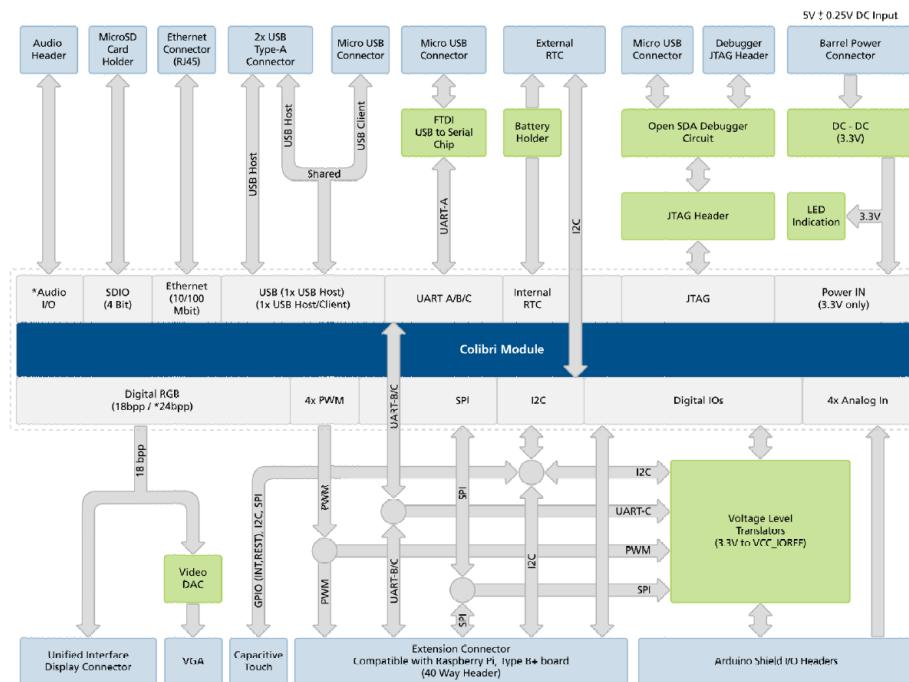


Figure 1.6: Aster board block diagram

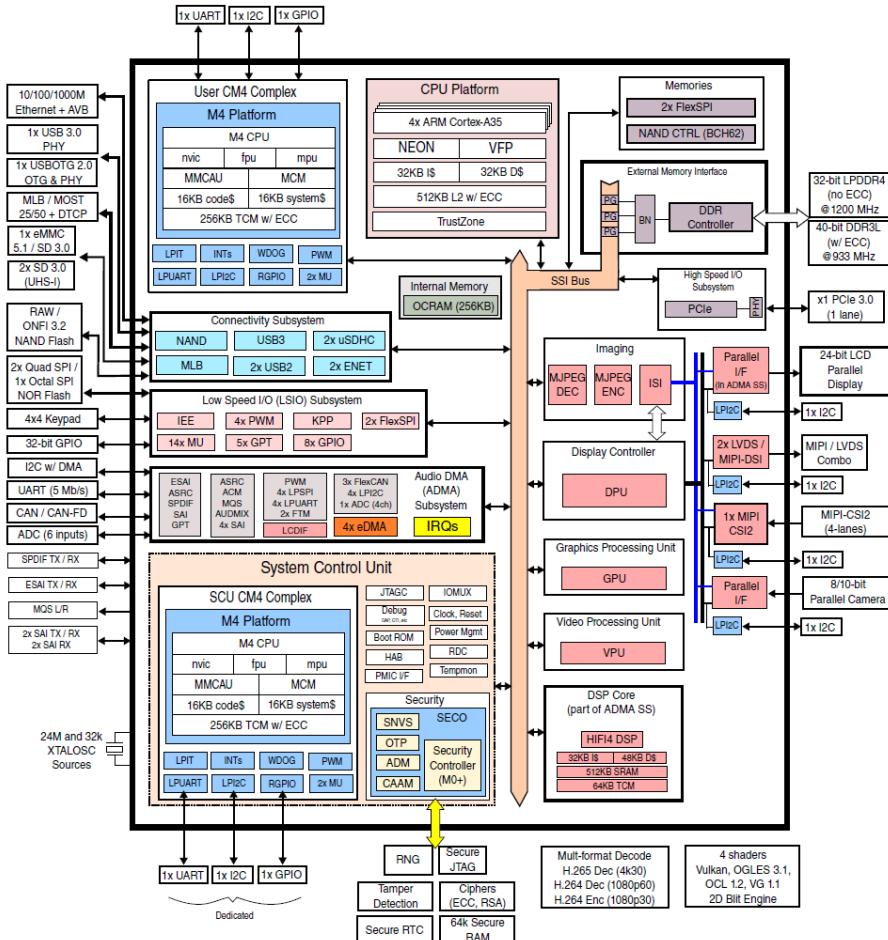


Figure 1.7: NXP i.MX8 simplified block diagram

GPU subsystems on the chip. The block diagram for the chip (Figure 1.7 is also quite impressively complex – compare this with Figure 1.1).

The good news is that you don't need to understand most of the documentation, and we'll supply clear pointers to where to look for the stuff you do need. One of our goals is that by the end of the course you should feel quite comfortable navigating this kind of documentation!

The OS you will write will actually run on the four principal cores on the i.MX, labeled “CPU Platform” at the top of Figure 1.7, which are ARM Cortex A35 64-bit processors implementing the ARMv8-A architecture (see the next section). The specifics of these cores can be found in the ARM Cortex-A35 Technical Reference Manual [13], but you should not need to refer to this much during the course -

Size	ARM terminology	Intel terminology	C type
8	byte	byte	char, uint8
16	halfword	word	short, uint16
32	word	doubleword	int, uint32
64	doubleword	quadword	long long, uint64

Table 1.1: ARM data sizes and terminology

the general ARM Architecture Reference Manual [12] is much more useful for the project.

We'll start with running on just one of these cores, and then bring up the others shortly after the mid-point of the course. There are actually many other cores on the i.MX, including some Cortex M4 cores, but we'll ignore them (unless you want to play with them in your spare time...).



Technical  
Details

## 1.6 The ARMv8-A Architecture

The code you'll be writing will be compiled for the ARMv8-A architecture, which is the latest general 64-bit version of the ARM architecture. The standard reference for this is the ARM Architecture Reference Manual [12], which you will need to refer to when reading and writing assembly language during the course.

We'll just give a brief overview of the architecture here. If you've dealt with a 32-bit or 64-bit RISC architecture before (like MIPS, SPARC, RISC-V, etc.) it will look mostly familiar, with a few famous (or infamous) ARM-specific features. If your background is more x86, you'll find it considerably simpler.

ARMv8-A deals with the usual different sizes of integer data types - 8, 16, 32, and 64 bits in size. Pointers are 64 bits long. We'll ignore floating point in this course. The different sizes of data items use different names from the Intel x86 conventions - see table 1.5.

The ARMv8-A instruction set is in most respects a classic RISC load/store architecture: ALU operations are register-register, all instruction encodings are 32 bits long, and direct branches are PC-relative with the (signed) offset encoded in the instruction.

The complete set of user-mode ARM general registers (not including system and control registers, which we'll see more of when we look into virtual memory) is shown in Figure 1.8.

There are 31 general-purpose registers, R0 to R30. Each can be accessed as a 64-bit quantity, in which case it is referred to as X0 to X30, or as a 32-bit value, referred to as W0 to W30.

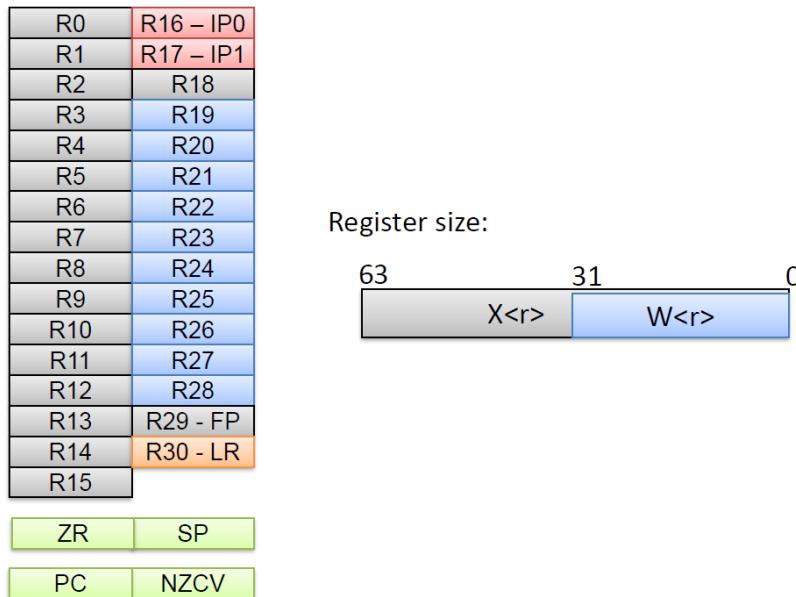


Figure 1.8: ARMv8-A registers at Exception Level 0

- R0 – R7 are used for procedure call arguments and results, and are therefore caller-save.
- R8 is used for indirect result locations.
- R9 – R15 are temporaries (also caller-save).
- R16 and R17 (IP0 and IP1) might be used by the linker between a procedure and any procedure that it calls, otherwise they can be used as temporary registers.
- X18 is the “platform” register – how it is used depends on the system. For example, it can be used to hold thread-local state.
- X19 – X28 are callee-save registers.
- X29 or FP is the frame pointer. It points to a 2x64-bit record in the next stack frame up the stack, though compilers may optimize out its usage for some stack frames, in which case it can be used as a general-purpose callee-save register.
- X30 or LR is the link register, which holds the return address from a procedure call.
- SP is the stack pointer, and is *not* a general-purpose register, but an additional

special register. The stack point should always be quadword-aligned.

- ZR is the “zero register” (which may not actually be a register) which can be used for some operands and always holds zero.

The program counter PC holds the address of the current instruction, and is also not a general-purpose register. Both SP and PC were general-purpose registers in the older ARMv7-A 32-bit architecture.

There is also a condition code register, which holds the Negative (N), Zero (Z), Carry (C), and oVerflow (V) flags. In user-mode code, it can be referred to as NZCV.

When a trap or exception happens, an ARMv8-A processor enters a different *processor mode* depending on what happened. In ARMv8-A, a processor mode is referred to as an Exception Level (EL), and there are four of them. EL0 is the lowest privilege level, and corresponds to user mode, while EL1, EL2, and EL3 correspond to higher levels of privilege. An exception always causes the same or higher exception level to be entered.

In each exception level, there is a different stack pointer (SP) register available, which greatly simplifies saving and restoring state. In addition, the condition codes (NZCV) are also automatically saved and restored.

The ARM Architecture Reference Manual has full details of the architecture; we’ll also look at particular aspects of it (such as the Memory Management Unit) later on in the course.



#### Technical Details

## 1.7 Grading Scheme

The course is not graded on a curve. We fix the grading scheme in advance, and grading is based on criteria rather than on how well the class as a whole performs. This means that you are only competing against the grading scheme, not against your classmates.

We assign equal weight to three different aspects of the course:

1. Whether you complete each milestone, demonstrate working functionality, and can explain it to the TAs
2. A set of tests we run on the final code, *after* the end of the course (meaning you have an opportunity to improve milestones as you go along, and as your subsequent work uncovers latent bugs in earlier functionality).
3. The final deliverable, which consists of the report (see Chapter 14) and the demonstration of the complete system showing the integration of the individual projects.

In the final deliverable, the score is weighted two-thirds to the report itself, and one-third to the demo and integration (see Figure 1.9)

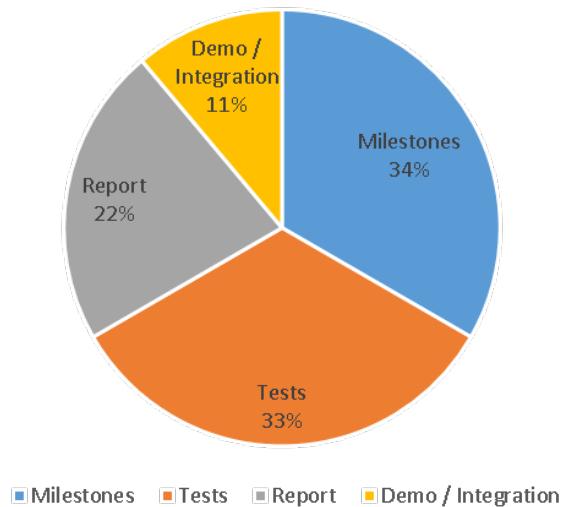


Figure 1.9: Grading scheme for the course

If each third of the course is given a score from zero to 3, this leads to final score from zero to 9.

The pass mark for the course is 3 out of 9 on this scale, and corresponds to an ETH grade of 4.00.

Scoring 8.5 or higher in the course will result in an ETH grade of 6.00 (and much kudos). You might want to consider a PhD in Systems.

## 1.8 Previous grade distributions

For comparison, the course was run in Autumn 2017 (with some minor differences) with the same grading scheme. 22 students completed the course, in 4 teams of 4 and 2 teams of 3 each.

The score in each section of the course is shown in Table 1.2. In this table, each column is calculated as if the score for each of the four components is out of 10, in other words, it does not take into account the different weighting in grading scheme. That year, nobody failed the course (and the minimum score of zero for the demonstration was something of an outlier), although some students did drop out.

However, it is also clear from the table that the highest *variance* in score is associated with the report, demonstration, and the final integration of the individual projects into a complete system.

Intuitively, this makes sense since the milestone and test suite parts of the course



*Background*

	Min	Max	Median	Mean	Std. Dev.
Milestones	6.14	9.86	8.07	8.04	1.40
Tests	4.71	8.14	7.04	6.78	0.95
Report	2.00	10.00	7.00	6.73	2.72
Demo / Integration	0.00	10.00	8.00	7.23	2.75

Table 1.2: Score distributions from Autumn 2017 (each score out of 10)

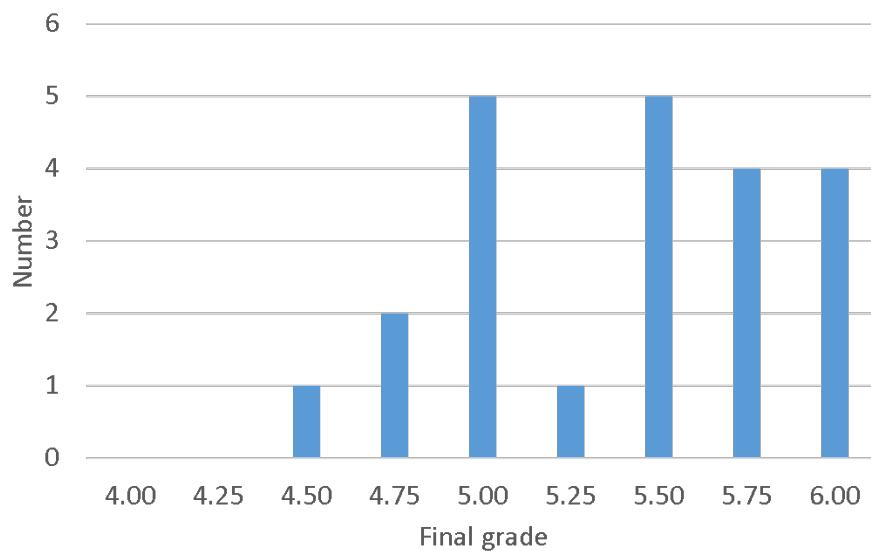


Figure 1.10: Final grade distribution for the course in 2017

are fairly bi-modal: either the milestone is delivered or it isn't, and either the test passes or fails. There is much more scope for differentiation in the report and demonstration, albeit only contributing to a third of the score in total.

For completeness, the distribution of final grades is shown in Figure 1.10.



### Commentary

## 1.9 Hints and Tips

Finally, before you start on the course, here are a few hints and tips to help you succeed and, hopefully, have fun.

Firstly, in this course you'll be judged on the design of your code. Does it work reliably? Does it handle corner cases, error conditions, unexpected events, invalid inputs, etc? An operating system is a program (or set of programs) that runs for a long time, essentially indefinitely. Anything that leaks memory, for example, is

going to fail sooner or later. Also, have you thought about issues not explicit in the milestones, but important to a real OS? Not all of the requirements or criteria for an OS can be well-specified in advance, since they depend on particular other design choices you are expected to make when building your system. We're looking for you to use common-sense in system design. You will be graded on (and rewarded for!) issues you can think of and deal with that are not explicit in the project milestones.

Secondly, you will be judged on the quality of your report. Tools like Doxygen document lines of code, not the design of a system or the thought that went into it. Don't submit generated documentation in place of a report – this is likely to result in a very low grade. Instead, describe the choices you made (and didn't make), and why. Discuss the tradeoffs. Mention the difficulties and challenges, and how you overcame them. Talk about your mistakes. Show you understand how to build a system.

A good way to create a really good (and interesting!) report is to write it as you go along, documenting how you are thinking at each stage and what you find out. Feel free to include performance experiments you run during the project to test out ideas.

Try to show the depth of your understanding: Design principles you follow, ideas you borrow or are inspired by. Explain why techniques work, when they work (or not), and what the tradeoffs are in any given (sub)problem. What factors affect those tradeoffs? How might this change if (for example) hardware changes in the future? Indeed, how has it changed so far?

Lastly, experience shows that this course is a lot of work, but also a lot of fun. It is important not to fall behind, and we don't want you to fall behind. The team is here to help, either in person during the consulting sessions, or via the support mailing list, or with ad-hoc meetings if you need them. If your team is struggling, *ask for help*. Conversely, if you're good and it's all going well, it's tempting to be clever and cool and implement extra functionality not in the milestones (like playing music using the audio hardware). Resist this temptation until you're confident with everything else! Get the required work done before you start freestyling.

Good luck, and have fun!



## **Part I**

### **Individual early milestones**



# Chapter 2

## Getting started

Having got the preliminaries out of the way, it's time to start dealing with a machine and some code.

This is the first milestone in the course. The goal here is to get you familiar with the build environment for Barrelyfish on the Toradex board, and also do some very simple device programming. At this stage it's pretty low-level; we'll get on to more deep OS concepts later in the course.

### 2.1 Check you've got the hardware

You will need several pieces of hardware from us for this course.

The first is the Toradex board itself. It consists of the Colibri module and the Aster carrier board. Please take great care of it – it's quite fragile. Always transport it in its box, protected by bubble wrap and the anti-static charge bag. Also, be gentle when plugging and unplugging connectors. You should not need to unplug cables very often. In particular, you should never need to take the module and carrier board apart.

For you to be able to quickly reset the board we provide you with a button attached to two jumper cables. If it's not connected to the board, connect it to JP1 (see Figure 1.5).

You are also going to get two micro-USB cables: One plugs into connector X4 which is attached to the USB UART and powers the Toradex board from the USB port on your computer. The UART appears as a device on the host machine and allows you to send and receive console input and output between the Toradex board and a terminal window on your host machine. In addition, the USB UART has some pins that *might* be wired such that the board can be reset over the USB cable, in which case it can be done under software control.



Project  
Instructions

The other one is used to connect to X10 which is the USB client port and is used to boot the Toradex board from your computer by downloading the OS image into memory.

First, connect the UART to your computer. We would like to emphasize here to be gentle when connecting the cables. Verify that you are connecting the cable in the right orientation. If you're tailing `/var/log/syslog` on your computer (`tail -f` is very useful in these situations), or you immediately run `dmesg`, you should see something like this:

```

1 [44174.940502] usb 1-8.3: new full-speed USB device number 65 using
    xhci_hcd
2 [44175.047309] usb 1-8.3: New USB device found, idVendor=0403,
    idProduct=6001
3 [44175.047314] usb 1-8.3: New USB device strings: Mfr=1, Product=2,
    SerialNumber=3
4 [44175.047317] usb 1-8.3: Product: FT232R USB UART
5 [44175.047320] usb 1-8.3: Manufacturer: FTDI
6 [44175.047323] usb 1-8.3: SerialNumber: A1069C3Z
7 [44175.050713] ftdi_sio 1-8.3:1.0: FTDI USB Serial Device converter
    detected
8 [44175.050792] usb 1-8.3: Detected FT232RL
9 [44175.051301] usb 1-8.3: FTDI USB Serial Device converter now
    attached to ttyUSB0

```

As you can see, the Linux host spotted the new USB device, identified it as a serial port converter, and made it available as device `/dev/ttyUSB0`. The exact device file name might be different on your system, but in this chapter we'll assume it's `/dev/ttyUSB0`.

Other than your computer detecting the new device, you should see at least LED1 (the one in the opposite corner from connector X4) lighting up to indicate that the Toradex board is powered.

If you can't see this output from `dmesg` or in `/var/log/syslog` or LED1 doesn't light up, there may be a problem with your USB cable or the Toradex board itself.

Otherwise, let's now plug the other USB cable into the Toradex board and your computer. You should see output like this in `/var/log/syslog` or from `dmesg`:

```

1 [31719.297583] usb 1-8.4: new high-speed USB device number 7 using
    xhci_hcd
2 [31719.398371] usb 1-8.4: New USB device found, idVendor=0525,
    idProduct=4026
3 [31719.398376] usb 1-8.4: New USB device strings: Mfr=1, Product=2,
    SerialNumber=3
4 [31719.398379] usb 1-8.4: Product: USB download gadget
5 [31719.398382] usb 1-8.4: Manufacturer: FSL
6 [31719.398384] usb 1-8.4: SerialNumber: 06410717

```

If you see something like this, there's a good chance you have a working Toradex board!

Note: The power supplied via USB is limited, and thus you should avoid connecting more devices to the board – especially when using an unpowered USB hub. Moreover, some laptops may not provide enough power when running on battery, and

may have designated ports with a higher power capability. It's best to connect both cables either directly to your computer, or to use a powered USB hub, and make sure your laptop is connected to a power source.

## 2.2 Check you've got the toolchain

In this course you will be compiling and linking OS code for the Toradex board, booting the machine, and monitoring the output via the console. The software we use runs on Linux, and consists of:

- The GCC C compiler, configured to compile code for ARMv8-A processors using the Linux ABI. We assume whatever GCC version is up-to-date for Ubuntu 20.04 LTS. Other versions may work, but we can only provide support for this one.
- The Glorious Glasgow Haskell Compilation System (GHC), again using the version current for Ubuntu 20.04 LTS. Some of the tools you will use to build the OS are written in Haskell, and we use GHC to compile them.
- picocom, a terminal emulation program for interacting over a serial line.
- git, for checking code out of the course repository.
- Standard GNU/Linux tools like bash, awk, make, etc.
- Additional Haskell and C libraries for the host machine.
- QEMU, for running the OS in a machine emulator.

If you are using one of the ETH Lab machines, everything should be already installed. If you prefer to use your own machine (which many students do), you will probably have to install some packages.

You are, of course, welcome to install these tools on whatever machine you like for use in the course, and many people do so successfully. On Ubuntu 20.04 LTS running the following commands should be sufficient to get (almost) all the tools setup:

```
$ sudo apt-get install build-essential bison flex ghc libghc-src-exts-dev \
    libghc-ghc-paths-dev libghc-parsec3-dev libghc-random-dev libghc-
    ghc-mtl-dev libghc-async-dev picocom cabal-install freebsd-glue \
    libelf-freebsd-dev git gcc-aarch64-linux-gnu g++-aarch64-linux-gnu \
    qemu-efi-aarch64 qemu-system-arm qemu-utils python3 parted \
    mtools
```

The only thing missing is an additional Haskell library that is not in the Ubuntu 20.04 LTS package libraries. On the ETH Lab computers we provide this library on a network share and you can tell the Haskell compiler about it by setting an environment variable:



Project  
Instructions

```
$ export GHC_PACKAGE_PATH="/net5/sg.fs.inf.ethz.ch/export/sg/nfs3/\
netos/teaching/aos/ghc/package.conf.d:"
```

We also provide a script in the handout that you can source to automate this (where BFAOS contains the path to the handout):

```
$ . ${BFAOS}/lab_env.sh
```

Note that you will have to do this again when you open a new terminal. On other machines you can install the library with the following commands:

```
$ cabal update
$ cabal install bytestring-trie
```

*Do not run these commands on the ETH lab machines.* The hackage package list and the library will be downloaded into your home directory which will quickly exceed your storage quota.

This is the only configuration we fully support. For any other host machines we can help to some extent with getting things working “natively” on your laptop or desktop machine, but there are simply too many variations for us to support them all. Ultimately, if you choose to use an environment other than the lab machines or Ubuntu 20.04 LTS, you’re on your own.

To test that the tool chain is (probably) installed correctly, you can try running the key programs:

```
$ ghc --version
$ aarch64-linux-gnu-gcc --version
$ picocom -h
```

On Ubuntu 20.04 LTS, you should see GHC version 8.6.5 and GCC 9.3.0 (or something close). If your GHC version is much different from this (in either direction), you’ll likely have trouble, and you might like to set up an Ubuntu 20.04 LTS VM or dual boot configuration.

## Docker Container

There is also a way to compile Barreelfish inside a Docker image. See the Bash script `tools/bfdocker.sh` in the source tree once you cloned the repository. The Docker image contains an Ubuntu 20.04 LTS repository and can be used build Barreelfish and the other tools required to boot the Toradex board. Note, booting Barreelfish from inside the Docker image is currently not possible with this image. To get this to work, one would need to map the USB device into the Docker container, something that would be theoretically possible.

## 2.3 Check out the project code

Now that you have your development machine set up, the next step is to check out the project repository. First, you need to know the git url of this repository. The rest of this book will assume that this address is available in the AOSREPO environment variable.

Now, clone the AOS Barrelyfish repository to somewhere convenient in your home directory. The first handout (milestone 0) is available on the `milestone0` branch which you can check out automatically when cloning the repository:

```
$ git clone -b milestone0 $AOSREPO bf_aos
$ cd bf_aos
```

You should now have a directory `bf_aos` in your home directory with the repo and associated files in it. The rest of this book will use the environment variable `BFAOS` as a placeholder when referring to your AOS source directory, and likewise we refer to `BFBUILD` as the build directory.

## 2.4 Creating a build directory and Makefile

We'll now build the image with which we will boot the Toradex board. First, create a build directory – this is where any files you compile will end up. In Barrelyfish, this is strongly separated from the source tree, which is the repository you just checked out above.

```
$ BFBUILD=${HOME}/build_milestone_0
$ mkdir -p ${BFBUILD}
```

You can create the build directory anywhere you like, but for clarity here we'll assume it's also in your home directory. While it is technically possible to put your build directory in the source tree (e.g., the build system ignores the build directory), this could lead to issues. Thus, for reasons that will become clear, it's not a good idea to put your build directory inside the source tree.

On some machines (such as those in labs), your home directory might be mounted over the network via NFS or SMBFS. If this is the case, you might instead want to put your build tree on the local hard disk (sometimes mounted under `/local`), as building will then be a *lot* faster. If you're using `/local` it's polite to create a directory with your user name and put your stuff in there. For example:

```
$ BFBUILD=/local/heidi/build_milestone_0
$ mkdir -p ${BFBUILD}
```

On the lab machines you might not have access to `/local`. In this case `/tmp` is another option. Note that `/tmp` is often backed by a RAM file system that isn't



*Project  
Instructions*



*Project  
Instructions*

persisted across reboots. The advantage is that it's even faster than the local hard disk.

Barrelfish is built using `make`. To create a `Makefile`, build it as follows (assuming both repo and build directories are in your home directory):

```
$ cd ${BFBUILD}
$ ${BFAOS}/hake/hake.sh -s ${BFAOS}
```

You should see a lot of messages on the screen, and eventually success. There should now be a file called `Makefile` in the top level of your build directory. You can look at this if you like, but be warned that it's very big and contains a lot of targets – this may cause autocomplete to be slow.

In the build directory, you can now run `make help` to see a menu with some useful targets.

```
1 -----
2 This is the 'help' target for the Barrelfish Makefile. This
3 Makefile
4 has been generated by hake in the top-level directory of your build
5 tree.
6 See the Hake documentation for more information.
7
8 This Makefile contains build targets for the following architectures
9 :
10 armv8
11
12 To change configuration options, edit the Config.hs file in the hake
13 subdirectory of this directory and run 'make Makefile'.
14
15 Useful make targets:
16   - <file>:           any file which can be built in the Barrelfish
17     tree
18   - 'help':            show this help text
19   - 'help-platforms': show the platforms (file collections) which can
20     be built
21   - 'help-boot':       show the boot sequences which can be initiated
22   - 'help-tests':      show the available tests
23   - 'clean':           remove most generated files
24   - 'realclean':       remove all generated files (clears the build
25     tree)
26   - 'Documentation':  build all Technical Notes
27   - 'Makefile':         attempt to re-run hake
```



*Technical  
Details*

## 2.5 What happened? What's this Hake thing?

The build process for Barrelfish uses two steps.

Scattered through the source tree are files called `Hakefile`, which specify what files need to be built in the tree. In the first step, a program called `hake` [91] generates a `Makefile`. It does this by collecting all the `Hakefiles` together, resolving any filenames they refer to, and interpreting them as single, large datastructure in Haskell. This expression evaluates, in the context of `hake`, to a single, large `Makefile`.

In the second step, `make` can be invoked with this `Makefile` to build any generated file in the system, whether it's a tool or a Barrelyfish binary or boot image. Barrelyfish only uses *one* `Makefile` for the whole build tree, but this file can contain hundreds of thousands of lines. The great thing about `make` as a utility is that it can handle these kind of size without breaking a sweat.

We go through this step in Barrelyfish so that we can build lots of programs, libraries, utilities, build tools, and kernels for a collection of different processor architectures and different platforms from the same source tree *at the same time and in the same build tree*.

Effectively, we treat `Makefile` syntax as an assembly language for the build system, and a full language (embedded in Haskell) as a high-level way to express what we want to build. In this way we get all the power of a full programming language in expressing configurations but still the efficiency and speed of using `make` for building things.

## 2.6 Test the boot tool

Now we'll test whether you can download some code from your host to the Toradex board. To do this we use a tool called [UUU](#) (Universal Update Utility). We also provide you with scripts that orchestrate the process.

One of the scripts needs a USB interface for Python. If you are using your own machine, make sure it is installed. On Ubuntu 20.04 LTS it is available in the package `python3-usb`.

Download UUU and make sure it is on your search path i.e.

```
$ wget -P $HOME/bin https://github.com/NXPmicro/mfgtools/releases/
      download/uuu_1.4.165/uuu
$ chmod +x $HOME/bin/uuu
$ export PATH="$HOME/bin:$PATH"
```

Note that by default Ubuntu 20.04 LTS adds `$HOME/bin` to your search path if it exists at login.

We give you a test image to see if the scripts, UUU, the cables, the Toradex board, and your computer are all correctly configured and working together.

You also may want to run the following commands to install a custom udev rule in `/etc/udev/rules.d/60-colibri.rules`, which allows any user on the system to access the Toradex board's UART and client port:

```
$ cat | sudo tee /etc/udev/rules.d/60-colibri.rules > /dev/null << EOF
# USB OTG
SUBSYSTEM=="usb", ATTR{idVendor}=="0525", ATTR{idProduct}=="\
        4026", MODE="0666"
# Serial USB device
```



Project  
Instructions

```
SUBSYSTEM=="usb", ATTR{idVendor}=="0403", ATTR{idProduct}=="\
6001", MODE=="0666"
# Serial TTY device
SUBSYSTEM=="tty", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="\
6001", MODE=="0666"
EOF
$ sudo udevadm control --reload && sudo udevadm trigger
```

Note, in case this didn't work, you may have a board with a product id. You can ask the `uuu` tool to generate the corresponding udev rules for you using the command:

---

```
uuu -udev
```

---

Now, fire up another terminal window, and inside it run `picocom` to monitor output on the serial port:

---

```
$ picocom -b 115200 -f n /dev/ttyUSB0
```

---

Note that `/dev/ttyUSB0` is the device node that we saw from `dmesg` when we connected the USB UART – if your machine assigned the cable a different device name, you should give this one to `picocom` instead.

`picocom` should start up and listen on the new serial port. You can quit `picocom` by using the key sequence `Ctrl-a Ctrl-x`.

Now, we'll use our orchestration script called `bf-boot.sh` which in turn uses `UUU` to send a compiled image to the Toradex board. The test image we've given you will print something to the console, and also flash an LED on the Toradex board.

Now press the reset button on the board. Note, this may cause the device `/dev/ttyUSB0` to disappear, and terminate `picocom`. If that happens, simply re-run the `picocom` command with the parameters above again. Then run the following in the first terminal window (not the one running `picocom`):

---

```
$ ${BFAOS}/tools/imx8x/bf-boot.sh --bf ${BFAOS}/\
milestone0_test_image
```

---

If you've done everything right, you should see something like the following output from `bf-boot.sh`:

---

```
$ ./tools/imx8x/bf-boot.sh --bf ./milestone0_test_image

Resetting board

Creating temp dir

Copying auxiliary files to /tmp/tmp.YwhHBm9z3K

Copying Barreelfish image to /tmp/tmp.YwhHBm9z3K
```

---

```

Booting board
uuu (Universal Update Utility) for nxp imx chips -- libuuu_1.4.72-0\
g8e9e189

Success 1 Failure 0

3:213 6/ 6 [Done] ] FB: done

```

– and, more importantly, something like the following output in picocom:

```

1 U-Boot 2020.04-5.1.0+git.0a26a04408ca (Dec 28 2020 - 14:02:46 +0000)
2
3 CPU:   NXP i.MX8QXP RevB A35 at 1200 MHz at 57C
4
5 DRAM:  2 GiB
6 MMC:   FSL_SDHC: 0, FSL_SDHC: 1
7 Loading Environment from MMC... OK
8 In:    serial
9 Out:   serial
10 Err:   serial
11 Model: Toradex Colibri iMX8 QuadXPlus 2GB Wi-Fi / BT IT V1.0B, Serial#
      06494292
12
13 BuildInfo:
14   - SCFW 732e719a, SECO-FW 376e3c15, IMX-MKIMAGE 6745ccdc, ATF
15   - U-Boot 2020.04-5.1.0+git.0a26a04408ca
16
17 flash target is MMC:0
18 Net:   eth0: ethernet@5b040000 [PRIME]
19 Fastboot: Normal
20 Normal Boot
21 Hit any key to stop autoboot: 0
22 Starting download of 503 bytes
23
24 downloading of 503 bytes finished
25 Starting download of 13928808 bytes
26 .
27 downloading of 13928808 bytes finished
28 ## Executing script at 83200000
29 Loading DeviceTree: imx8qxp-colibri-aster.dtb
30 120412 bytes read in 19 ms (6 MiB/s)
31 51 bytes read in 16 ms (2.9 KiB/s)
32 Applying Overlay: colibri-imx8x-aster_led6_overlay.dtbo
33 984 bytes read in 22 ms (43 KiB/s)
34 Scanning disk usdhc@5b010000.blk ...
35 Scanning disk usdhc@5b020000.blk ...
36 Found 5 disks
37 Blob is at: 0xFC654D90
38 Magic: 12345678FEDCBA90
39 Relocating ELF FC64C000 FC666D90 1
40 Relocating ELF FC646000 FC667D90 12
41 Relocated CPU driver entry point is FFFF0000FC646B80, stack at
      FC642000
42 Relocating multiboot info: FC635000
43 FC635008: tag 1:32
44 FC635028: tag 19:56
45   before 1000:1D27
46   after FC638000:FC638D27
47 FC635060: tag 19:48
48   before 2000:A36F
49   after FC639000:FC64136F
50 Allocating 512 L1 tables (2097152B)
51 L1 tables start at 0xFC434000

```

```

52 Terminating boot services and jumping to image at 0xFC64C000
53 Core data pointer is FC433000
54 Memory map at FC66CA20, key: 0, descriptor version: 1
55 Got 22 memory map entries of 40B (880B).
56 Type          PStart        PEnd        Size      Attributes
57 available    0000000080200000 00000000FC432FFF 2033868kB 8
58 BF core data 00000000FC433000 00000000FC433FFF 4kB 8
59 BF page table 00000000FC434000 00000000FC633FFF 2048kB 8
60 BF page table 00000000FC634000 00000000FC634FFF 4kB 8
61 BF multiboot 00000000FC635000 00000000FC637FFF 12kB 8
62 BF module   00000000FC638000 00000000FC641FFF 40kB 8
63 BF stack    00000000FC642000 00000000FC645FFF 16kB 8
64 BF code     00000000FC646000 00000000FC64BFFF 24kB 8
65 BF code     00000000FC64C000 00000000FC64CFFF 4kB 8
66 LD code     00000000FC64D000 00000000FC670FFF 144kB 8
67 reserved   00000000FC671000 00000000FC671FFF 4kB 8
68 reserved   00000000FC672000 00000000FC672FFF 4kB 8
69 reserved   00000000FC673000 00000000FC673FFF 4kB 8
70 reserved   00000000FC674000 00000000FC674FFF 4kB 8
71 reserved   00000000FC675000 00000000FC675FFF 4kB 8
72 reserved   00000000FC676000 00000000FC676FFF 4kB 8
73 reserved   00000000FC677000 00000000FC677FFF 4kB 8
74 reserved   00000000FC678000 00000000FC678FFF 4kB 8
75 reserved   00000000FC679000 00000000FC679FFF 4kB 8
76 LD data    00000000FC67A000 00000000FFF17FFF 57976kB 8
77 RS code    00000000FFF18000 00000000FFF18FFF 4kB 8
78 LD data    00000000FFF19000 00000000FFFFFF 924kB 8
79 AOS M0 CPU driver starting on ARMv8 (BSP)
80 kernel 0: ARMv8-A: Global data at 0xfffff000080200000
81 kernel 0: ARMv8-A: Kernel stack at 0xfffff0000fc642000.. 0
           xffff0000fc645ff0
82 kernel 0: ARMv8-A: Kernel first byte at 0xfffff0000fc646000
83 kernel 0: ARMv8-A: Exception vectors (VBAR_EL1): 0xfffff0000fc646000
84 kernel 0: ARMv8-A: 4 cores in system
85 kernel 0: ARMv8-A: Welcome to AOS!

```

You should also see LED4 flashing . If so, congratulations! We're getting there. The rest of this milestone consists of reproducing the output of the boot image you have just tested, and understanding how it all works.



*Technical  
Details*

## 2.7 Understanding Booting

What just happened?

The process for booting any machine is complex, and usually very specific to a particular piece of hardware. For example, the way we have just booted our test kernel on the Toradex board is very different from the way that a PC or an iPhone boots.

The basic principles, however, are almost always the same:

1. One core starts up executing out of Read-Only Memory (ROM).
2. The core puts the rest of the machine hardware into a known state.
3. It copies a program from storage or from across the network to a well-known location in memory.
4. It then jumps to a well-known address in that program (the 'entry point').

In the past, this was the entire boot process. However, those were more innocent times, and today the last two steps of this process often happen multiple times in sequence when a machine boots.

The iMX8X is quite a complex SoC with multiple CPU cores of different types. It is not uncommon for a platform like this to have a dedicated processor core that bootstraps the rest of the system – the Reference Manual for the chip [84] refers to this as the System Control Unit (SCU).

## SCU bootstrap process

When the SoC is reset, the SCU executes code from its on-chip ROM which loads a so-called “boot container” from internal flash memory. This boot container contains the firmware for the SCU and a bootstrap image for the first of the main A35 cores on the SoC.

The SCU firmware first initializes some platform components like the DRAM controller and then loads the image for the A35 core into RAM. It then powers up the first A35 core, which starts executing this image. Take a look at chapter 5 of the iMX8X Reference Manual [84] for more details.

## Cortex-A35 bootstrap process

The A35 starts executing the “ARM Trusted Firmware” at Exception Level 3 (EL3), which is the most privileged level of the processor.

This firmware itself then executes a boot loader called U-Boot [37] at the lower privilege level of EL2.

U-Boot is highly flexible and configurable. In the case of the Toradex board, it loads the image you built over the USB On-The-Go (USB-OTG) port using Fastboot on the host side.

This image contains three components:

1. “Hagfish light”, a bootloader for Barreelfish which expects to run in a Unified Extensible Firmware Interface (UEFI) [104] environment,
2. Binary index/relocation info for Barreelfish boot driver and CPU driver, and
3. Several concatenated binaries

U-Boot then sets up a UEFI environment and runs Hagfish Light, which relocates two binaries: the boot driver and the CPU driver. Hagfish Light then builds kernel page tables, creates a multiboot image which indexes the remaining binaries in the boot image along with other important memory regions, and then jumps to the boot driver passing it a `struct armv8_core_data` which describes the core that the CPU driver needs to boot on.

The boot driver is the last step of the process before the CPU driver itself. It configures the translation regime for EL1 and EL0, installs kernel page tables for EL1, and finally drops the exception level to EL1 and jumps into the CPU driver.

At this point, the Toradex board is finally executing code you have compiled into the CPU driver. The good news is that you shouldn't need to worry too much about what has happened up to this point. Nevertheless, it's good to know what has occurred to set up the environment that your kernel code is finally running in.

In later milestones, you will add user-level tasks to your boot image, which will appear in the output of `readelf` as sections with names like `.elf.binary_name`. This enables us to load different programs from the multiboot header without needing a full file system up.



*Project  
Instructions*

## 2.8 Test the Virtual Machine

Running Barrelyfish inside a virtual machine can provide some additional features that could turn out to be useful for debugging your code, in particular one can attach `gdb` to Qemu.

We are now going to test the virtual machine setup of your development environment. This is an optional step. You are free to skip this section if you were given an actual hardware board for the course. If so, remember that we will test your submitted code on the actual hardware, and not Qemu.

You can test your Qemu setup using the following command:

```
qemu-system-aarch64 -m 2G -cpu cortex-a57 -M virt,virtualization=on,\  
gic-version=3 -smp 4 -nographic -bios tools/hagfish/QEMU_EFI.fd -\  
device virtio-blk-device,drive=image -drive if=none,id=image,file=\  
milestone_0_qemu_test_image,format=raw -nographic
```

In contrast to the Toradex board, you will see the output printed directly in the terminal you've typed the command in. Indeed, you will see quite some output, that should end in the following print:

```
1 AOS M0 CPU driver starting on ARMv8 (BSP)  
2 kernel 0: ARMv8-A: Global data at 0xffff000040000000  
3 kernel 0: ARMv8-A: Kernel stack at 0xffff0001380ed000 .. 0  
4 kernel 0: ARMv8-A: Kernel first byte at 0xffff0001380e6000  
5 kernel 0: ARMv8-A: Exception vectors (VBAR_EL1): 0  
6 kernel 0: ARMv8-A: 0 cores in system  
7 kernel 0: ARMv8-A: Welcome to AOS!
```

To terminate Qemu, press `CTRL-A x`.



## 2.9 What just happened?

The `qemu-system-aarch64` command creates a new emulated platform following the Arm virtual machine base platform (`-M virt`). It configures the platform with four Arm Cortex-A57 cores and two Gigabytes of memory. It then tells the emulator which BIOS to use – here we want the EFI environment that is already pre-compiled in the tree. Qemu will load this file into memory at address 0 and starts executing it. In fact, you can just pass and load any binary there and have Qemu execute it, but be aware: this is read-only memory.

The actual Barrelyfish image is supplied as a virtual disk. During the boot process you will quickly see the EFI shell that then reads the contents of the disk that contains a bootscript (`startup.nsh`), the Hagfish bootloader, and the Barrelyfish image.

The bootscript tells EFI Shell which commands to execute. In our case, this is simple to execute Hagfish with the supplied configuration file that lists the files of the Barrelyfish image.

Hagfish.efi hagfish.cfg

Hagfish then reads this configuration file, copies the modules specified in it into memory, creates the page tables, and loads the bootdriver.

## 2.10 Building your own kernel

Now for some code. You'll build your own image that we can boot on the Toradex board. For this, we'll start with some code that you obtained when you checked out the milestone 0 branch from the repository earlier.

This tree is a subset of the main Barrelyfish source tree, but contains most of the files needed to build a single, bootable program for the Toradex board. There's also a lot of extra code that isn't relevant to this milestone, but it does give you an idea of the complexity of even a research OS like Barrelyfish.

The kernel in Barrelyfish is called the CPU driver, and sits in the source directory `kernel`. In this directory are some files that are portable across all kinds of machines, and others that are specific to particular instruction set architectures, processor models, or platforms. The latter are in the various directories under `kernel/arch`.

To start, take a look at `kernel/arch/armv8/boot/boot_entry.S`. This is the first code which is executed after the EFI application that we use as a shim between U-Boot and our kernel. For the first core that comes up, only the code under the label `boot_entry_esp` is executed. EFI leaves us in a relatively luxurious environment (as far as boot environments go), meaning that we are already executing on a stack and can jump straight to code produced with a C compiler. You'll find this code in `kernel/arch/armv8/boot/boot_generic.c`. In here we configure the MMU such



Project  
Instructions

that the actual CPU driver can execute and then jump to the CPU driver, dropping to EL1 (ARM speak for kernel mode).

If all went well, we find ourselves executing the function `arch_init` in the file `kernel/arch/armv8/init.c`. Right now it tries to print some status information, and then call a function named `aos_blink_led()`.

The first step is to see if we can build this trivial kernel.

Remember, if you lose the Makefile in your build tree you can always recreate it by rerunning `hake` inside your build directory:

```
$ ${BFAOS}/hake/hake.sh -s ${BFAOS}
```

Next, build the kernel:

```
$ make armv8_aos_m0_image.elf
$ ls -l armv8_aos_m0_image.elf
```

Recall, the command `make help` will give you a list of all the possible targets you can build, and more specifically `make help-platforms` to get a list of all supported platforms that can be build. For example, to build the image above execute `make AOS_M0`.

Platforms supported by this Makefile. Use 'make_<platform_name>': (these are the platforms available with your architecture choices) Documentation: Documentation <b>for</b> Barrels <b>AOS_M0:</b> AOS Milestone 0 Image <b>AOS_M0_QEMU:</b> ARM Virtual Machine Board <b>in</b> QEMU
---

Likewise, `make help-boot` will give you a list of possible boot targets. Basically, this is a wrapper around the commands stated above. Thus, to boot your newly compiled kernel on the board execute `make usbboot_aos_m0`.

Boot instructions supported by this Makefile. Use 'make_<boot_name>' : (these are the targets available with your architecture choices)  <b>qemu_aos_m0:</b> Boot QEMU <b>in</b> 64-bit ARM mode emulating a ARM Virtual \ Machine using UEFI <b>usbboot_aos_m0:</b> Boot AOS M0 on a Colibri board, over a <b>local</b> USB cable
--

The build process should work with no compilation errors. If so, congratulations! You've built a Barrels CPU driver that does nothing. You can, of course, boot this on your Toradex board using the following command, but you won't see any output

beyond the boot shim, since the code to write to the UART and to blink the LED is missing.

```
$ ${BFAOS}/tools/imx8x/bf-boot.sh --bf armv8_aos_m0_image.efd
```

Your task for this milestone will be to fix this, but don't worry we'll walk you through the process.



## 2.11 Console output

First, we want some output. The USB UART that you connected to your host machine is itself connected to another UART that's part of the i.MX 8X chip which is at the heart of the Toradex board. We'll write a very simple driver for it that can output a character to the device.

Note, the Qemu virtual machine is different from the i.MX 8X chip. It uses another UART device that is at another address. We will provide some more information at the end of this section.

The UART is documented in the Technical Reference Manual (TRM) for the i.MX 8X, Section 16.13 [84].

Here's what you need to know:

1. There are actually 4 identical UART devices on the Toradex board SoC, but we are interested in LPUART3, which is the one connected to the USB UART. The registers for this device start at address 0x5A090000 in physical memory (see table 2-6 in Section 2.2.3 in the TRM).
2. There are instructions on how to program the UART in Section 16.13.2 of the manual, but for now we *won't* be doing anything this complicated. In particular, we won't be using interrupts, DMA, or FIFOs. Instead, we just want to get a single character out of the serial line for debugging purposes.
3. Furthermore, we know that the UART will already have been initialised by the bootloader (that's how the bootloader printed something to your picocom window) by the time you get to `arch_init()`. You just need to figure out how to send characters.
4. To send a character to the serial line, your code first needs to wait until the UART is ready to send a character. This happens when the transmit data register is empty, indicated by bit 23 (named TDRE) of the Status Register (STAT) becoming set (i.e. equal to 1). This register is described in Section 12.5.3.1.7 of the manual, and for our UART you can see that it sits at address 0x5A090014.
5. To actually send a character once the UART is ready, we write the character into the Data Register (DATA), which is a address 0x5A09001C and is described in Section 12.5.3.1.9 of the manual (though there is not a lot to say about it!).

One more thing you need to know, is that you need to translate physical addresses to kernel virtual addresses. We provide you with a function `local_phys_to_mem()` that does this.

You now know all you need to know to write a character to the serial port, so you should go ahead and fill in the body of the simple function called `aos_serial_putchar()`, in `kernel/arch/armv8/aos_m0.c` which does this. It should loop waiting for TDRE to be 1, and then write the character into DATA.

Test this out by adding a line to `arch_init()` to call your new function with an argument of 42. Rebuild the kernel, boot it, and you should see an asterisk (\*) as the last thing in the output on picocom.

Once you've got this, of course, you can try it with other characters. However, the kernel has a minimal C library which includes `printf()`, which in turn calls `aos_serial_putchar()`, so you should be able to now use this. This makes debugging subsequent code much, much easier.

**QEMU UART.** The emulated platform uses a PrimeCell UART (PL011). This has a different register layout. The UART is located at address 0x9000000. This is for your information only, and we don't require you to implement the UART for the Qemu emulated platform.

## 2.12 A Simple Interpreter/Shell

Next we implement a small little command line interpreter which runs over the serial port. Note, to see what you are actually typing, you will need to implement some form of echoing of characters back to the user. Implement a few commands, like `help` and `hello`.

For this you will need to read input characters from the UART and store them in a buffer – so you will need to implement `serial_get_char()`.

By looking at chapter 12.5 of the Soc manual, you can see that the DATA register is actually two different registers: if you write to it (which is what you've been doing so far to output a character), it acts as the transmit data register. However, by reading from it, you get the first unread character received by the UART over the serial line.

You can tell whether the UART has received any data by testing the RDRF bit of the STAT register.

When you recognize that the return key has been pressed, you evaluate the buffer and try to recognize the commands. Note, you will need to handle a new input after you've handled the command!

Keep it simple, there is no need to handle arbitrary input sizes, backspace or auto-completion as of now (see [chapter 9](#) for the corresponding milestone implementing a proper shell).



Project  
Instructions

## 2.13 Flash the LED

Now that you can write strings to the console port, the final exercise for this milestone is to flash LED4 (the one nearest the SD card reader) on the Toradex board.

This LED is controlled by one of the General-Purpose Input/Output (GPIO) lines. Take a look at the Aster Carrier Board Datasheet [103] in Section 3.3.2, and you'll see that it's connected to SODIMM pin 104 of the Colibri module. If you now chase



Project  
Instructions

the signal to the Colibri iMX8X Datasheet [102] and look at the table in Section 4.4.1 you'll find that it's connected to pin QSPIOB\_SS0\_B of the SoC which can be configured to be one of four signals from hardware on the SoC. ALT4 is pin 23 on the third GPIO block on the SoC (GPIO3) and U-Boot already took care of configuring the pad multiplexers accordingly. Table 2-4 in Section 2.2.1 of the i.MX 8X TRM [84], tells us that this GPIO block is at address 0x5D0B0000.

Programming the GPIO pins is described in Section 18.4 of the TRM, and they can do a lot of different things (they're *general purpose*). However, all we need to turn the LED on or off is two registers: the Direction Register, or GDIR, and the Data Register, or DR. These registers have a bit for each of the 32 I/O pins controlled by the GPIO3 block, and you need to set the correct GDIR bit for the LED to configure the pin as an output, and then set or clear the correct DR bit to turn the LED on or off.

Your remaining task is to flash the LED on and off about once per second, by implementing the function `aos_blink_leds()`, again in `kernel/arch/armv8/aos_m0.c`.



*Extra  
Challenge*

## 2.14 System Shutdown/Reset

Ok, so far you've implemented a simple command line interpreter. Let's add two more commands to it: `shutdown` and `reboot`.

On Arm platforms that implement the Arm Power State Coordination Interface (PSCI) [14], this can be done by invoking the right PSCI function. The functions of interest are outlined in sections 5.10 and 5.11, and its general use in section 4.4 "System shutdown, reset and suspend".

All right, so far we know which PSCI functions we need to call. Next, is to figure out *how* to call them. This information is present in section 5.2 "Arguments and return values in PSCI". It outlines how to invoke the PSCI function, and how to pass arguments and obtain return values – the latter is obviously not so important for us at the moment.

On Toradex board, the PSCI function is invoked by calling the `smc` instruction. See the SMC calling conventions for more details [17]. The Firmware will then catch the invocation and emulate the appropriate behavior.

Now, we've got everything: we know which PSCI functions to call, and how to do the calling using the `smc` instruction. Let's implement the `aos_shut_down()` and `aos_reset()` functions. The expected behavior is as follows:

- `aos_shutdown()` triggers a shutdown and Toradex board will turn off, i.e., you are back in the terminal.
- `aos_reboot()` triggers a reset: you will see Toradex board reboot, and you will be back in the kernel after a few seconds.

Hint: you will need to use *inline assembly* to invoke the `smc` instruction. Make sure that the arguments are passed in the right registers.

## 2.15 Milestone 0 Summary

Milestone tasks:

1. Build and boot an image on the Toradex board.
2. Demonstrate console input/output from a booted image.
3. Demonstrate flashing the LED on and off.
4. Demonstrate a mini command interpreter/shell handling of a “hello” command.



Project  
Instructions

Extra challenge:

1. Handle Shutdown and Reboot through PSCI.

Submission:

You should submit your code as specified on the course website before the specified deadline, and be prepared to demonstrate your code in the marking session.

Assessment:

1. You will need to demonstrate that you can build and boot an image.
2. Explain your code for serial output/input and the minishell with “hello” command.
3. Explain your code for LED control to the tutor.
4. Describe the boot process to the tutor.



# Chapter 3

## Memory management and capabilities

### 3.1 The first real milestone

In the first true milestone, you will implement the infrastructure for managing (physical) memory. You will also see and use *capabilities* for the first time as a mechanism to identify regions of physical memory, and create page tables to map physical memory frames into your program's virtual address space.

From here on, we'll give you the CPU driver (kernel) that you will be using, as well as a very basic initial user-space process (called, as in Unix, `init`) which you'll add to during the project. We also give you the skeleton of a library called `libmm` which you will flesh out to provide memory allocation.

This milestone is the first step towards a (hopefully) complete memory system at the end of this course. The goal for this milestone is to perform memory *allocation* and basic memory *mapping* functionality within the `init` address space. Note that in the forthcoming weeks you will continuously refine and extend your memory management system with more functionality. Therefore, *it is important that you write correct and understandable code to form a solid basis for future assignments*.

### 3.2 Memory management in general

Memory management is a huge topic. We'll assume you already know the basics of memory management from any undergraduate OS class: virtual vs. physical memory, physical frame allocation, MMUs, page faults, address space layout, and demand paging and swapping.



Commentary



Background

In a monolithic kernel like Unix, all of this is handled in the kernel. Alternative designs like microkernels and exokernels, including Barrelyfish, push quite a bit of this functionality out of the kernel into user-space code, in some combination of libraries or server processes.

Microkernels like L4 [77], Mach [92], Chorus [1], and Spring [64] trapped page faults in the kernel but then reflected them up to other processes which carried out the actual page fault handling. This was done on a per-region basis, so each area of virtual memory was associated with some paging server (via something that Mach called a “memory object”, which was supposed to abstract the physical resources backing the region). Memory objects could be shared between different processes and mapped differently in different address spaces.

This abstraction is quite powerful: basically, what happens when a page fault happens is entirely dependent on the code in the user-level pager. In addition to traditional demand-paging to disk, for example, you can mark a memory object as *copy-on-write*, and then the pager creates copies of page frames in response to any attempt to mutate the data. You can share pages between different machines, and use page faults to make sure they look consistent [82]. You can lazily allocate (and zero) pages to allocate frames only when the process touches them. You can also map things which aren’t memory but device registers.

Most of these things can be done (and have been done) in Unix or Linux by adding more code to the kernel, but the difference in microkernels is, firstly, that this is highly *extensible* – it’s all user code – and it’s also *isolated* – if a user-level pager crashes, there’s a good chance the rest of the OS can continue quite happily. Some research systems like SPIN [25] and VINO [41] allow users to download code into the kernel to provide similar flexibility; and the isolation is provided by some way of verifying the code for safety before the kernel installs it.

A different approach is to allow a process to build its own address space, and reflect page faults back to *it*. Some variant of this approach is used by the Aegis Exokernel [42], Nemesis [52], seL4 [40], and the V++ Cache Kernel [56], among others.

However, moving functionality out of the kernel in any way raises an important question: if user-space processes can manipulate virtual address spaces, how can we make sure that one user’s program can’t manipulate another’s address space and memory (unless it is explicitly intended)?

This essentially comes down to:

1. A *naming* problem: how to refer to regions of physical memory, address spaces, other processes, etc.
2. An *access control* problem: how to ensure that only authorized processes (or users, or actors, or ...) can perform a given operation on a given object.

## 3.3 What are capabilities?

*Capabilities* are a very general, and a very old, idea in Computer Science. They crop up in all kinds of places (for example, Java object references are capabilities), and in contrast to other ways of organizing programs or systems, they solve the naming and access control problem referred to above *at the same time*.

One thing needs to be said first, to prevent confusion: you may be familiar with the concept of capabilities in Linux. The problem is, as will become clear, Linux capabilities aren't really capabilities as any computer scientist would understand them. It's best to pretend for the moment that you have never heard of Linux capabilities.

To explain capabilities a bit more, and how they are used for memory management in systems like Barrelyfish, we'll first need to take a short detour through the subject of access control.

### 3.3.1 Access control in general

*Access control* is the problem of specifying, and enforcing, which *subjects* (or *principals*) can perform particular actions on particular *objects* in an operating system.

For example, subjects might be human users (or processes they start), or particular program binaries. Objects might be files, directories, processes, areas of memory, sockets, etc. Actions generally depend on the type of the object, and include reading, writing, deleting, sending a Unix signal, etc.

Abstractly, access control can be thought of as a matrix, which represents all possible combinations of operations in the system. Each row of the matrix represents a different subject, and each column represents a different object. Each entry in the matrix contains a list of permissible actions.

Naturally, this matrix is pretty large – too large to represent naively in an operating system, since adding a new subject requires adding a new entry for every object in the system, and vice versa. The matrix is also a highly dynamic structure with frequent changes.

Instead, operating systems compress this matrix, sometimes limiting what combinations of rights can be represented, so that changes can be performed quickly, and the storage required is not too much – ideally, space should scale no more than linearly with the number of objects and/or subjects.

For any access control scheme, we can ask a number of questions about what functionality it provides:

**Propagation** : Can a subject grant access to an object to another subject?

**Restriction** : Can a subject propagate a subset of its own rights?

**Revocation** : Can access, once granted, be revoked?

**Amplification** : Can an unprivileged subject perform restricted operations?



Background

**How is object accessibility determined** : How does one determine which subjects have access to a particular object? How can one determine if an object is accessible by any subject (for example, for garbage collection)?

**How is a subject's protection domain determined** : How does one determine which objects are accessible to a particular subject?

The two most common schemes for representing the access control matrix are:

1. **Access control lists** or ACLs, which store (presumably) sparse columns of the matrix with each object
2. **Capabilities** which store a row of the matrix with each subject

Access control lists are implemented by most mainstream systems. An ACL is associated with each object, and encodes the list of subjects which have access to the object along with the rights they have on that object (i.e. which operations they can carry out).

Access control list schemes vary, but in general have the following properties:

**Propagation** : ACLs can't directly express the right to propagate rights, but it can be encoded as a "meta-right" – a right on another object (implicit or explicit) which "owns" or "contains" this one. For example, the owner of a file in Unix may chmod the file, which is effectively modifying the file's meta-information.

**Restriction and Revocation** are similarly meta-rights.

**Amplification** is a protected invocation right (eg. setuid properties on an object in Unix)

**Accessibility** is trivial to determine: it is explicit in the ACL

**Protection domain** , in contrast, is hard (if not impossible) to determine: at a bare minimum, it involves examining every ACL in the system.

In practice, ACL schemes do not mention every individual subject explicitly. Instead, ACLs are usually condensed via *groups* or *classes* of subjects. To complicate things further, they sometimes encode *negative* rights (e.g. "everyone in group users can read this file *except* user davidc"). Finally, the ACL itself is sometimes implicit: in the UNIX process hierarchy a limited number of processes can send a signal to a given PID.

As hinted above, classic Unix file privileges are a (restricted) ACL representation, where each ACL lists the rights of a single subject (the file owner), a further group of subjects (the file's group), and a third implicit group of subjects (everyone else) – see Figure 3.1. The superuser also has a fourth, implicit entry.

Windows has a much more sophisticated access control list implementation, which allows arbitrary numbers of both individual subjects and groups. There have also been various attempts to implement similar functionality in Unix-like operating systems, for example the POSIX ACL facility in Linux (try "`man 5 acl`").

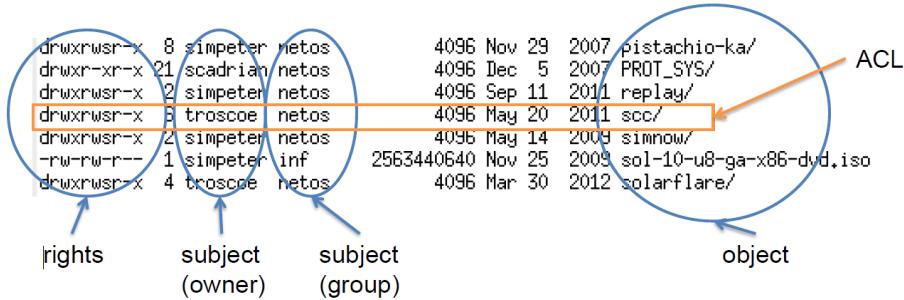


Figure 3.1: Access control lists in classical Unix

### 3.3.2 Capabilities

While an ACL is associated with an object and lists subjects, capabilities are associated with a subject and refer to objects.

A capability can be thought of as a “key” or “license”. It is an unforgeable token which grants authority. Possession of a capability for an object gives the holder the right to perform certain operations on the object. Nothing more is required.

If you’re not familiar with capability systems already, there are a number of subtle points here worth mentioning.

First, a capability combines naming an object and authorizing operations on that object. This is not the case with ACLs: to operate on a file, you first need the name for the file, and then the ACL is checked. A capability itself functions as a name (this is why object references in Java resemble capabilities).

Secondly, capabilities decouple *authorization* from *authentication*. When an ACL check is performed to see if you can read a file, the system needs to verify that you are who you say you are (*authentication*) in order to look you up in the ACL (*authorization*). With capabilities, the system doesn’t care who you are: simply possessing a capability is enough to grant you access.

Capabilities provide fine-grained access control: it is easy to provide access to specific subjects, and it is easy to delegate permissions to others in a controlled manner. Note that to be correct, any capability representation must protect capabilities against forgery.

Capability systems have the following characteristics:

**Propagation** is trivial: in general capabilities can be freely copied or transferred between subjects (though this can be restricted in some systems, permitting a form of *mandatory access control*)

**Restriction** requires creating a new (derived) capability from an existing one, where the new capability has more limited rights to the same object. Most capability

systems support some form of restriction.

**Revocation** requires invalidation of capabilities for an object held by all subjects in the system, and may be difficult – revocation is generally cited as the principle drawback of capability systems.

**Amplification** of rights can be handled by capability systems in several ways, for example having an “invocation capability”: an object holds its own capabilities not accessible outside it, and can be asked to perform operations on behalf of holders of the invocation capability.

**Accessibility** determination is basically the same problem as revocation: to revoke a capability entails finding all subjects that hold the capability.

**Protection domains**, in contrast, are trivial to determine for subjects: the capability list (or CList) for a subject is *precisely* its protection domain – this fact is one reason security researchers have often found approaches based on capabilities attractive.

Capability systems have a very long history in computing, going back at least as far as the “descriptors” used in the Burroughs B5000 in 1961, and current research using capabilities includes the CHERI processor work at Cambridge [108]. Hank Levy’s book from 1984 gives a good history up to the early 1980s, though omits one of the most important capability-based operating systems, KeyKOS [54], perhaps because it was an industrial project (used commercially for several decades) but not widely known at the time.

### 3.3.3 The confused deputy

The power of the coupling of naming and authorization inside a single object was neatly illustrated by Norm Hardy (one of the designers of KeyKOS) in a problem he called “the confused deputy” [53], which he claims is based on a true story involving an OS that was not Unix, but essentially similar in the way it performed protection. The company operating the machine, Tymshare (which later built KeyKOS) made its money in the 1970s by renting out its mainframe and charging users for the CPU time and other resources they used. These days, this is called “cloud computing”<sup>1</sup>.

The following is a simplified version of the actual problem, translated into Unix terminology, but you are urged to read the original paper – it’s entertainingly and well written, and analyses the problem in much more detail than here.

Suppose a C compiler collects billing information and has to record this in a file somewhere (e.g. `/var/spool/billing`) for the billing software to process later (Figure 3.2).

When the user types:

```
$ cc -o ~/program program.c
```

---

<sup>1</sup>More specifically, Platform-as-a-Service or PaaS.

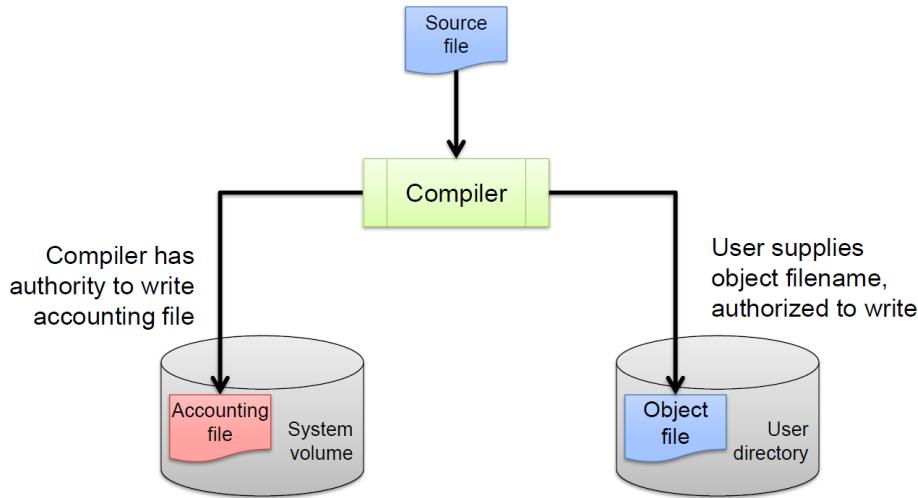


Figure 3.2: The confused deputy problem

- the compiler reads the user's file `program.c`, compiles it into machine code, writes the binary into `program` in the user's home directory. It also writes information to the billing file `/var/spool/billing`.

Obviously we can't allow users access to `/var/spool/billing`. Instead, we can give the `compiler` binary write access to this file, with something like a Unix setuid bit. Now only the compiler can write to the billing file, but the users can run the compiler to compile programs.

However, bad things happen when an unprivileged user types:

```
$ cc -o /var/spool/billing program.c
```

The problem is that the compiler has two masters, the user (on whose behalf it is supposed to write a binary file), and the system (on whose behalf it is supposed to write billing information). The compiler acquires blanket authority from both – hence the term “confused deputy”.

Hardy points out that if you use capabilities instead of file names and access control lists, the problem does not arise (Figure 3.3).

In this case, the compiler is always clear whose authority it is acting under, since that authority is tied to the very name of the resource it is trying to access. It accesses the output binary file via a capability supplied by the user, so this can never be a system file (unless the user already has access to such a file), and accesses the billing database via a different capability supplied by the system (and attached to the compiler beforehand). No confusion arises.

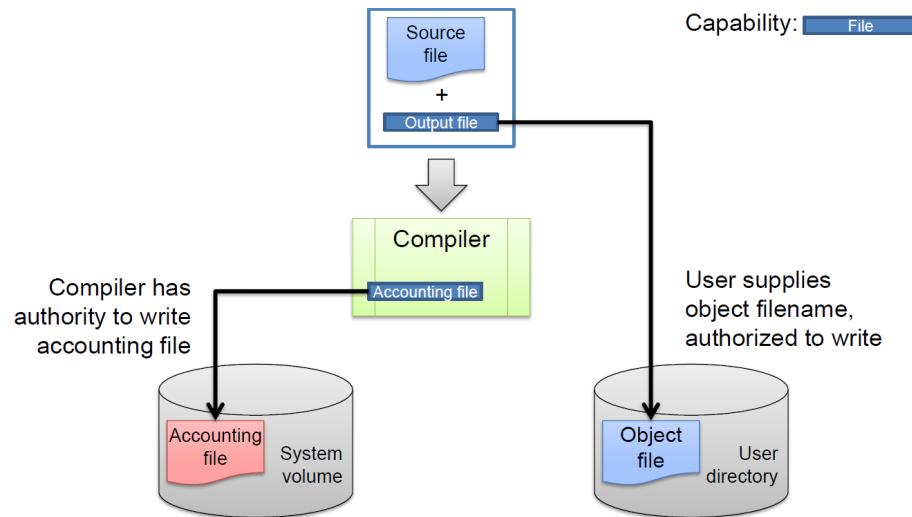


Figure 3.3: Solving the confused deputy problem

### 3.3.4 Linux capabilities?

It should now be clear why Linux capabilities (those described by `man 7 capabilities`) are not really capabilities in any true sense: they can't be refined, passed around, revoked, etc., and they don't name resources per se (they still grant blanket authority, just less of it). They are really wrappers around a centralized authority (the kernel) which keeps track of which processes can perform certain privileged operations – in this sense, they are just as close to access control lists.

However, Linux *does* have things that look a lot more like capabilities, and are indeed sometimes used as such: file descriptors. A file descriptor cannot be forged, names something (such as an open file, or a shared memory segment, or a socket), and confers on a process holding a file descriptor the right to perform various operations on the object it points to. Indeed, Linux and some other Unix-derived operating systems even allow file descriptors to be passed from one process to another, over a Unix domain socket using `sendmsg()`.



Background

## 3.4 Implementing capabilities

So much for capabilities as abstract concepts. How are they actually implemented? A real capability system has to be able to use capabilities as references for objects in the system, without allowing anyone in the system (other than a minimal Trusted Computing Base or TCB) to create a capability out of nothing.

Three main approaches have been used traditionally: *tagged* capabilities, *sparse*

capabilities, and *partitioned* capabilities.

### 3.4.1 Tagged capabilities

Tagged capabilities are implemented by hardware, which keeps track of which locations in memory currently hold capabilities and which hold regular data. Each word of memory (or, more likely, each small region of memory big enough to hold a capability, e.g. 16 bytes) has an additional *tag* bit associated with it indicating what is stored there.

Capabilities can be loaded from memory with the tag set into special *capability registers*. Storing a capability from a capability register to memory causes the tag for that region of memory to be set. Storing data from a general-purpose register to memory that has the tag bit set will either fail, or cause the bit to be cleared depending on the design.

In this way, capabilities can be used and copied just like normal pointers, and indeed in many such systems they *are* how pointers are represented. The difference is that a pointer can never be created from an integer, as it can in most computers today.

The only way to set the tag bit on a memory region without storing a valid capability to it is to have sufficient privilege to forge capabilities. This might be limited to the kernel, something below the kernel, or in some cases the hardware at boot time.

With tagged capability systems, **propagation** is easy: it is simply done with capability register loads and stores. **Restriction** involves creating a new, weaker capability based on an existing one; many tagged capability systems provide machine instructions for this. **Revocation** is difficult: in principle it requires a scan of the whole memory to identify all copies of a capability. Determining **Accessibility** is also virtually impossible without looking at the whole content of the RAM.

Despite these downsides, the fact that tagged capabilities are highly convenient for programmers (they are simply pointers) has meant that a number of hardware systems have been built to use them, including the IBM System/38 (later AS/400, then iSeries, then System i) [58, 74] and the current work on CHERI at the University of Cambridge [108].

Tagged capabilities can also be extended outside of DRAM, though a challenge here is that stable storage devices typically do not implement tags. The IBM System/38 simulated them by restricting physical I/O to the low-level operating system, which stored an extra bit for every word on disk. When paging memory out to disk, the page was scanned and tags collected to be stored alongside the page itself. When paging in, the tags are reconstructed on the fly. This does lead to considerable I/O overhead, particularly since the OS needs to be involved in the data path.

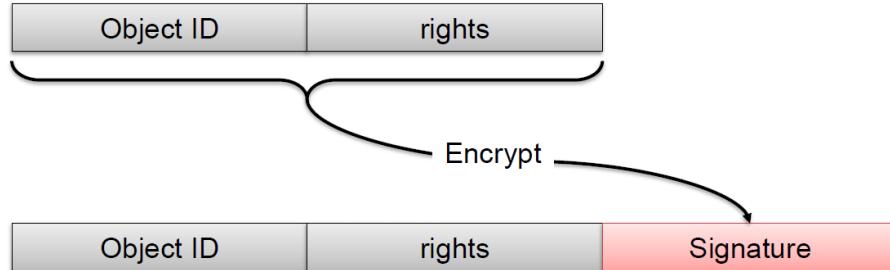


Figure 3.4: Encrypted sparse capabilities

### 3.4.2 Sparse capabilities

An alternative to marking capabilities in hardware using memory tags is to simply treat them as any other data, but make it almost impossible to forge capabilities by making the set of *valid* capabilities a vanishingly small subset of the full capability space.

This might be done using encryption: if only the kernel can decrypt the bit pattern of a capability, it can be passed around as regular data and presented as an authorization token, but users cannot forge the capability since they do not possess the encryption key. Alternatively, the capability might simply be *signed* by the kernel with a secret key.

Figure 3.4 shows an example: the object identifier and the associated rights are signed by the TCB to create an encrypted sparse capability.

However, even simpler schemes are possible: *password capabilities* simply store a long(ish) random bit stream with the identifier and rights information; the kernel has a list of such “passwords” and compares them to those stored in a global object table when a user attempts to invoke an operation.

Sparse capabilities are therefore regular user-level data, and can be passed around much like any other bits (much like tagged capabilities, but without the hardware support). As with the latter, **propagation** is trivial and **restriction** requires help from the TCB. However, since sparse capabilities have to be validated by the OS at invocation time (either explicitly or implicitly), **revocation** can be performed by deleting entries in the global table, or (more clumsily) by changing encryption keys and maintaining lists of revoked capabilities. Determining **accessibility** is still difficult as a consequence of the ease of propagation.

Some nice tricks that can be achieved with tagged capabilities are not hard or impossible with sparse capabilities. For example, tagged architectures allow “full mediation”: by carefully choosing the capabilities allowed into and out of a user’s capability space, you can completely “virtualize” their environment without extra privilege.

However, in some cases, such as distributed capability systems, where capabilities must be sent over an untrusted channel like a local area network, sparse capabilities implemented using encryption may be the only design option available.

### 3.4.3 Partitioned capabilities

In partitioned capabilities, the kernel (or TCB, strictly speaking) ensures that memory used to store capabilities is always separated from that used by user processes to store data and code, for example by using the MMU or ensuring that capability memory is only accessible in kernel mode. The OS maintains the list of capabilities each user principal holds (the *clist*), and explicitly validates access when performing any privileged operation (eg. mapping a page).

User code refers to capabilities using handles (indirect references) to the capabilities. A simplified form of this is exemplified by POSIX file descriptors, which function as capabilities. Each user process refers to objects by file descriptors, which are actually indexes into the process' open file table (analogous to a *clist*). The table is, itself, a list of pointers into the global open file descriptor table in Unix, which is where the real resources are held.

As a consequence, **propagation** in a partitioned capability system requires OS help, usually in the form of a system call to copy a capability between clients. Similarly, **restriction** is an invocation on the kernel to create new capability. **Revocation** is, for once, somewhat easier to achieve, since the kernel has (in principle) a list of all the copies of capabilities in the system, and can also maintain an index of these. **Accessibility** can be determined by in principle scanning all *clists*, but again an index can make this operation efficient.

Partitioned capabilities are used in a variety of capability systems past and present, including Hydra [111], Mach [2], KeyKOS [54], EROS [95], seL4 [40], and Barrelyfish.

Many of these systems use a more sophisticated mechanism for separating capability and data memory than the simple kernel-space approach used in POSIX file descriptors. In these systems, *all* regions of memory are referred to by capabilities. Regions have types, for example a “frame” type might be a page which can be used to back a virtual page in an address space and therefore store user data or code.

Capabilities are stored in areas of RAM with a type called a “CNode”. A CNode is divided into “slots”, each of which can hold a capability representation. CNodes are unreadable (and, more importantly, unwriteable!) from user space, but there is nothing otherwise special about them.

The clever part is that the type of an area of memory is entirely determined by the type of the capabilities that point to it – strictly speaking, memory itself isn't typed, but capabilities which refer to it are. In order to protect the integrity of the capability system – that is, to ensure that nobody can create a new capability from nothing – the system must ensure that there can never be an area of memory referred to by both a frame capability *and* a CNode capability. If there was, a user could write the

bits of capability into a frame, and then refer to the capability they just created by a slot number on the corresponding CNode.

Barrelfish uses a partitioned capability system for pretty much all objects in the system: the decision was taken early on to provide one general mechanism for tracking all operating system resources. This might not have been the best idea.

The disadvantages arise from generality: the mechanism must be fully general, and therefore at least as complex (and probably more so) than any single mechanism really needs to be. This in turn can lead to a potential performance hit for some operations, when compared to systems like Unix which use a different naming and authorization scheme for almost every type of kernel resource: files, open files, sockets, processes, threads, address spaces, signal handlers, user accounts, devices, etc. etc. etc.

However, the scheme comes with a lot of advantages as well. If you solve the problem once, everything works and, indeed, if you can formally prove the capability system is correct (as the seL4 team actually did [66]) you get a very high level of assurance that your system is secure at the lowest level – far more than Linux, Windows, iOS, Android, or MacOS can ever hope for.

Moreover, when you introduce a new set of resources (as we did recently with interrupt vectors and sources) you have a ready-made scheme for referring to them, which saves a lot of duplication of effort.

Finally, from a research perspective, it's highly instructive: we've gained a lot of insight into the general protection and naming problem for multicore hardware.

On balance, we think it was the right thing to do.



*Technical  
Details*

### 3.5 Capabilities in Barrelfish

All memory in Barrelfish (and some other system resources which do not occupy memory) is described using capabilities, in a design originally derived from seL4. Capabilities are typed, and capabilities can be **retyped** by users holding them according to certain rules and restrictions. Capabilities referring to memory regions can also be **split**, resulting 2 new capabilities of the same type, one for each half of the region. Some of the more important capability types in Barrelfish are shown in figure 3.5.

Retyping and splitting operations on capabilities provide an elegant solution to a key problem in operating system design and implementation: allocation and management of physical memory.

Most kernels dynamically allocate memory to hold kernel objects (such as process control blocks, socket buffers, etc.), for example Linux uses a “slab allocator” for this process. The kernel is constantly allocating and deallocating memory for a wide variety of purposes, and much as any large C program relies heavily on `malloc` and `free`. The problem is what the kernel should do when this runs out. The current

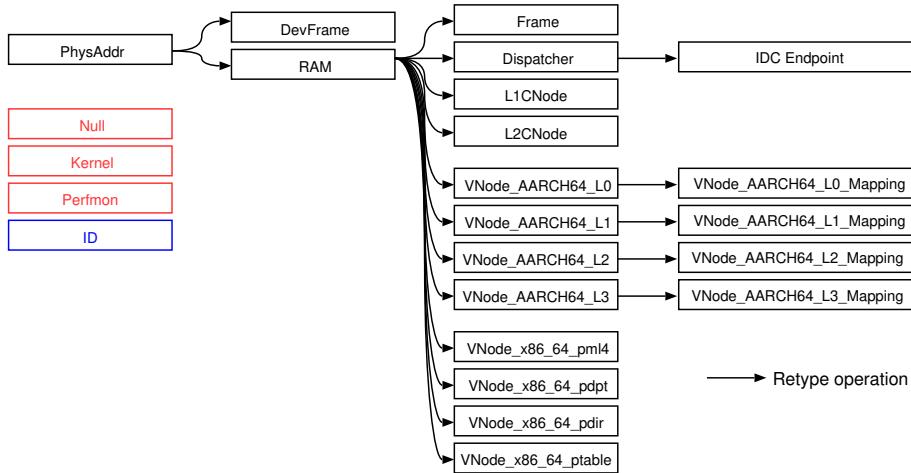


Figure 3.5: Some of the capability types in Barreelfish

solution in Linux is little more than “kill a random process and reclaim its memory”, which can be a problem for system stability.

In Barreelfish (and seL4), all kernel objects are actually allocated by user programs. For example, if a user process wants to create another process (or *dispatcher* in Barreelfish parlance), it has to get a capability to a DRAM area of the right size, retype this capability to type *Dispatcher*, and hand this to the kernel.

Note that it doesn’t ever get the right to read and write this memory – in order to do that, it would have to retype it instead to a *Frame* and then map it into its address space. Similarly, if a user process needs storage for its capabilities, it needs to retype some RAM it already has a capability for to type *CNode*. The process is illustrated in Figure 3.6.

The Barreelfish CPU driver *never* allocates any memory once it has booted. Consequently, it can never run out of memory. No memory leak in the system can cause the kernel to stop running.

Since a user in a partitioned capability system can’t manipulate capabilities directly, they need to refer to them somehow. In Barreelfish every capability resides in a slot in a *CNode*, so a pair (*CNode*, *slot*) would identify a capability. However, the *CNode* is another capability itself, so this doesn’t work.

Instead, each process in Barreelfish has a *CSpace* which is structured as a two-level table. There are actually two different *CNode* capability types - one for the first level of the table, and one for the second. Every process has, within its “dispatcher control block”, a pointer to the top-level or root *CNode* which the kernel can traverse (see Figure 3.7).

A capability reference in Barreelfish is a bit like a (32-bit) virtual address: the first

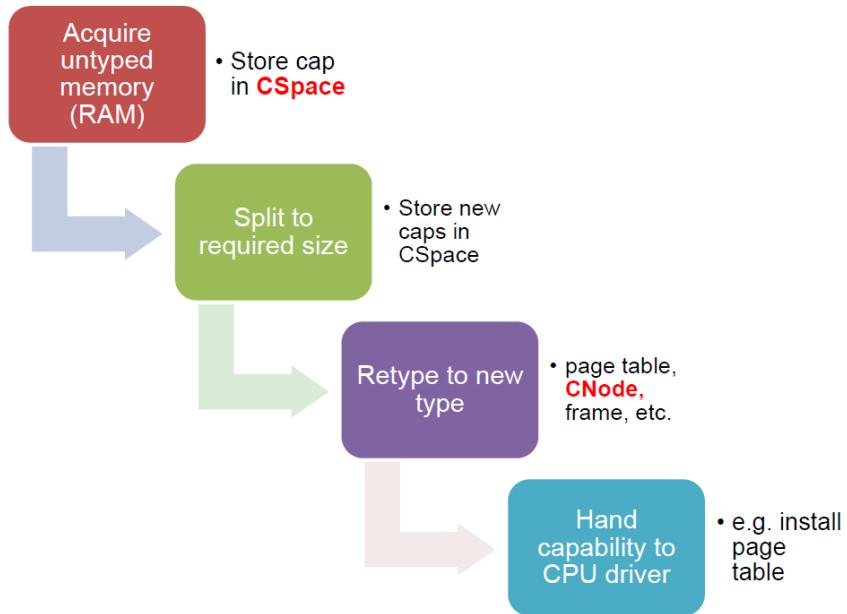


Figure 3.6: Capability retying

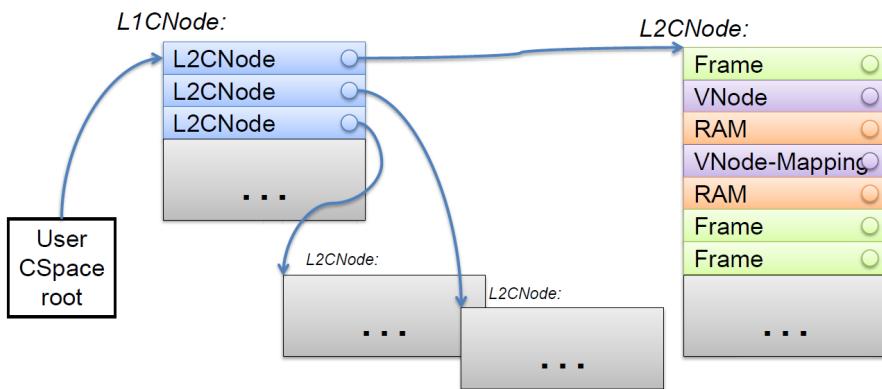


Figure 3.7: The CSpace

few bits refer to a slot in the L1CNode, and the next few refer to a slot in the CNode referred to by the capability in that L1 slot.

Each capability type in Barreelfish supports a set of operations as well as referring to areas of memory. You can think of capabilities as a bit like object references (or abstract data types). Most “system calls” are actually capability operations, and take a capability reference as the first argument.

## 3.6 Getting prepared

For this milestone, and for the rest of the course, you will get a full (though slightly simplified) Barreelfish CPU driver, a somewhat more complete init process, the libc C library, an initial version of the library OS libaos that you will use and extend, libmdb (which is used by the kernel to track capabilities), and a bunch of helper libraries you may find useful.

This will be the foundation of the OS you will build in the coming weeks. However, you might want to update it periodically from our git repo to pick up any bug fixes we introduce during the course.

The handout for the rest of the course is on the `main` branch of the handout repository. Note that from now on, we will refer to the git remote for the handout repository as `handout`<sup>2</sup>, as we assume that you are using another remote repository to keep track of your project progress. The branches for milestone 0 and the `main` branch don't have a common ancestor as they don't build on each other. So to avoid trouble, don't merge `main` into your branch for milestone 0. You can pull the new branch with the following commands:

```
$ git fetch handout
$ git checkout main
$ git push -u origin main
```

Note that this assumes that no branch called `main` existed in your repo beforehand. If you had your code for milestone 0 on a branch called `main`, it's probably a good idea to rename it first and then pull `main` from the handout:

```
$ git checkout main
$ git branch -m milestone0
$ git push -u origin milestone0
$ git fetch handout
$ git checkout -b main handout/main
$ git push -uf origin main
```

In this milestone, you will build the full Barreelfish CPU driver, as well as your first user-space program (init). Therefore, the commands have slightly changed. First, create a build directory. Recall, \$BFAOS is pointing to the source directory. Note that



Project  
Instructions

<sup>2</sup>You can rename a git remote by running `git remote rename <old> <new>`

if you're working on your own machine you can create the build directory anywhere you like):

```
$ BFBUILD=/path/to/build_milestone_1
$ mkdir -p ${BFBUILD}
$ cd ${BFBUILD}
$ ${BFAOS}/hake/hake.sh -s ${BFAOS} -a armv8
```

Next, build the tree. Recall, `make help-platforms` will provide you with the possible targets that can be build, here we want the following (check the man pages of `make` to figure out what `-j7` does):

```
$ make -j7 imx8x
```

Finally, boot the created image on the Toradex board. From `make help-boot` we get the following command:

```
$ make usbboot_imx8x
```

And make sure that you have access to the Toradex board console output:

```
$ picocom -b 115200 -f n /dev/ttyUSB0
```

An unmodified source tree should give you the following output:

```
1 spawn module: armv8/sbin/init
2 kernel 0: init page tables: I0=0xffff000080520000, I1=0
   xffff000080521000, I2=0xffff000080522000, I3=0xffff000080532000
3 load_init_image 0xffff0000fc457000 000808c7
4 kernel 0: ARMv8-A: init loaded with entry=0x4003c4 and GOT=0x457d60
5 kernel 0: ARMv8-A: Calling dispatch from arm_kernel_startup, entry
   point 0x4003c4
6 init.0.0: paging_init
7 ERROR: init.0 in bootstrap_thread() .../lib-aos/threads.c:1119
8 ERROR: error during libbarrelfish init
9 Failure: ( libaos) Failure in vspace_init() [
   LIB_ERR_VSPACE_INIT]
10 Failure: ( libaos) functionality not implemented yet [
   LIB_ERR_NOT_IMPLEMENTED]
11 init.0.0: libc exit NYI!
12 kernel 0: ARMv8-A: Timer interrupt received!
```



*Extra  
Challenge*

## 3.7 Bugs!

From this point on in the course, you'll be using the regular Barrelfish kernel (or CPU driver, as we call it). We'll offer extra points for every bug that you can find *and fix* in our code!



## 3.8 Error handling in Barrelfish

Barrelfish is written in C, and the C language does not provide any dedicated mechanism for exception handling. Instead, there are different conventions reporting an error from within a C function. In the C standard library, the return value in combination with a global variable is used to indicate an error condition. For example `fopen` can return a NULL pointer and, in this case of an error condition, the global variable `errno` is set to indicate the cause of the error.

In Barrelfish we rely entirely on the return value, which avoids the multi-threading issues with the global `errno` variable. Additionally, the Barrelfish error reporting conventions support *stacking* errors: an error code is actually a small stack of errors so that not only the outermost error is shown, but also the cause of that error (and the cause of that one and so on). Most of the error-related code is generated from a domain specific language we call *fugu*, and all the possible error conditions are defined in this language in the file `errors/errno.fugu`.

The error codes are grouped into domains and each domain has a prefix. For instance the system library for this course (`libaos`) errors have the prefix `LIB_ERR_` and in this domain there is an error defined `NOT_IMPLEMENTED`. The build system will invoke the *fugu* compiler to produce `build/<arch>/include/errors/errno.h`, which contains the integer values of all the error codes – for example, the error above would have an identifier of `LIB_ERR_NOT_IMPLEMENTED` – and the generated file `build/<arch>/errors/errno.c`, which contains textual representations of each of the errors.

The result is that your code can simply include `errors/errno.h` and use this as a return value in your function to indicate the error. The functions that interact with the error system are also defined in this header file. Two C types are used: `enum err_code` represents a single error, and `errval_t` represents a stack of errors and is, by convention, what each function returns. The stacking of errors works by bit packing multiple error values into a `errval_t` integer, hence there is a (small!) upper limit of stacked errors that can be reported.

To determine if the return code is not an error, the functions `err_is_ok()` and `err_is_fail()` should be used. If there is an error, the function `err_no` can be used to determine the last error on the top of the error stack. You can push new errors onto the stack using `err_push()`.

For debugging purposes, the macros defined in `barrelfish/debug.h` are handy, for example `DEBUG_ERR` will print the location of invocation plus the textual representation of the error. If the error is unrecoverable, `USER_PANIC_ERR` can be used to also abort the current domain, which makes it less likely to lose an error condition in a long output trace.

You should feel free to define your own errors or re-use existing codes if they fit your error case.



Technical  
Details

## 3.9 The Barrelyfish capability API

You will encounter the following types in this weeks assignment (see `capabilities/caps.hl` for a full list of Barrelyfish's capability types):

- `ObjType_RAM` (for untyped, physical memory)
- `ObjType_L1CNode` (for maintaining information about capabilities itself)
- `ObjType_L2CNode` (for maintaining information about capabilities itself)
- `ObjType_Frame` (for pages)
- `ObjType_VNode_AARCH64_10` (for ARMv8-A level 0 page tables)
- `ObjType_VNode_AARCH64_11` (for ARMv8-A level 1 page tables)
- `ObjType_VNode_AARCH64_12` (for ARMv8-A level 2 page tables)
- `ObjType_VNode_AARCH64_13` (for ARMv8-A level 3 page tables)

The two-level CSpace for a process is constructed on demand by retying RAM capabilities into L1CNode and L2CNode types. These CNode regions are only accessible by the CPU driver, however the user-space program is responsible for allocating them on the CPU driver's behalf. Note that L1CNodes have a dynamic size. Initially, they contain 256 slots holding L2CNode capabilities. If you ever run out of L1CNode slots you can use the function `root_cnode_resize()` to grow it. In contrast, L2CNodes have a fixed size and contain 256 slots.

There are also a number of functions that allow you to request new capabilities of these two types. You will likely encounter and use:

```

1 cnode_create_l1(struct capref *ret_dest, struct cnoderef *cnoderef)
2 cnode_create_l2(struct capref *ret_dest, struct cnoderef *cnoderef)

```

– to allocate level-one and level-two CNodes.

```

1 frame_alloc(struct capref *ret, size_t bytes, size_t *retsize)

```

– to allocate Frame capabilities that can be mapped into an address space, and

```

1 pt_alloc_l1(struct paging_state *st, struct capref *ret)
2 pt_alloc_l2(struct paging_state *st, struct capref *ret)
3 pt_alloc_l3(struct paging_state *st, struct capref *ret)

```

– to allocate capabilities for different levels of AARCH64 page tables.

All these functions will rely on the RAM allocator function:

```

1 ram_alloc(struct capref *ret, size_t bits);

```

- to allocate untyped, physical memory first and then retype that area into more and more specialized type by calling `cap_retype` on the respective capability. The invocation to retype a capability looks like this:

```

1 cap_retype(struct capref dest_start, struct capref src, gensize_t offset,
2           enum objtype new_type, gensize_t objsize, size_t count)

```

As you can see, the retype operation is fairly generic and can not only change the type of an existing capability (or region) but also modify its size, or work on just a part of the source capability. Note, in principle it is possible to create multiple new objects using a single call to `cap_retype` by passing an appropriate value to the `count` parameter. However, this requires a *range* of free slots, and thus may complicate tracking of free capability slots; so start by retyping only one new capability first.

The first part of this exercise is about implementing the necessary infrastructure for `ram_alloc` which will then give you the means to allocate all the other objects as well.

The `struct capref` type is what we use to reference a capability slot (and the capability in it) in user space. It uniquely identifies one address (i.e., one slot) in the process' capability space. It can be understood as a pointer to a region inside an L2CNode memory area that stores information about that particular capability.

You need to make sure that applications cannot construct arbitrary MMU page tables – instead, you need to ensure that an application can only put an entry in a page table if it refers to a physical frame to which it already has authority.

To enforce this, page tables are updated via the capability system. You invoke capabilities as if they were objects – essentially calling a function associated with the capability's type – and we use these invocations to (among other things) modify page table entries. The high level interface for writing a page table entry looks like this:

```

1 vnode_map(struct capref dest, struct capref src, capaddr_t slot,
2            uint64_t attr, uint64_t off, uint64_t pte_count,
3            struct capref mapping)

```

The `slot` argument gives the page table entry to modify, `attr` are the mapping attributes (you probably want `VREGION_FLAGS_READ_WRITE`), and `off` is an offset into the `src` memory region. The `pte_count` parameter is there to indicate the number of pages you'd like to map if you're mapping a Frame capability of more than one page (i.e., 4KiB) in size. In the last parameter, `mapping`, you need to supply an empty capability slot, which you can allocate using `slot_alloc`. This capability slot will be filled with a *mapping* capability that can be used to refer to the mapping that is created by `vnode_map`. Conceptually, the *mapping* is a subtype of a Frame capability and thus allows us to find all instances where a Frame has been mapped by walking the derivation tree in the kernel.

You'll notice that `vnode_map()` looks a bit generic, even though you'll only be using it for a couple of cases. One reason is that we use the same function to map L1 page tables in the L0 page table, L2 page tables in L1 page tables etc. and also pages in the L3 page tables. You will see the other reason for this later in the course – Barrelyfish can actually construct page tables for *any* processor architecture this way, even on a different architecture.

Barrelyfish stores the capability to the root level (L0) page table at a well-known location in the capability space, namely slot zero of the page `cnode`. You can create a `struct capref` to that location by hand as follows:

```

1 struct capref root_pagetable = {
2     .cnode = cnode_page,
3     .slot = 0,
4 };

```

– and use that `struct capref` as the destination parameter for `vnode_map()`. `cnode_page` is defined in `lib-aos/capabilities.c`.

Note, in order to create mappings in the page table, you will need to remember where in the CSpace the capabilities representing the page tables reside, and how they are mapped.

Certain capability operations are potentially heavyweight, as they may require tracking down other related capabilities, and thus they can't be finished within bounded time. Barrelyfish solves this issue by breaking up the operation into a series of single steps. This requires the corresponding user-level domain (i.e., the *monitor*). However, the monitor doesn't exist yet, as we will be building parts of this in a later milestone (see [chapter 13](#)). For now, `cap_revoke` will fail, as well as `cap_delete` on the *last* copy on a dispatcher or CNode capability.



*Project  
Instructions*

## 3.10 Task 1: a physical frame allocator

In the first part of this exercise, you will implement `libmm`, a library to manage memory (i.e., RAM capabilities). The library will form the basis for dynamically provisioning memory resources to various applications running on your system. In particular, you need to implement the various stub functions in `lib/mm/mm.c`.

On a high-level, there are two parts to this task:

1. Decide *which* memory to allocate. For this you will need to find some unallocated memory with the help of your data structure.
2. Perform the corresponding *capability operation*. Here will need to derive the capability to be returned to the client, and update the data structure to reflect the new allocation.

As you've just seen, the Barrelyfish model delegates memory management to user-space applications (e.g., `init` in our case). Initially, the kernel will query the under-

lying hardware to figure out which address ranges are backed by physical memory. From this information it will construct a series of RAM capabilities which it provides to `init` on start-up. Study the `initialize_ram_alloc` function in the file `usr/init/mem_alloc.c`. You will find that it walks through all the initial RAM capabilities as provided by the CPU driver and tries to add them to your memory manager using `mm_add`. Note, you may get an arbitrary number of memory regions that are added with `mm_add`. Moreover, you can use the function `cap_direct_identify` to obtain the address and size of the memory region the capability refers to.

Afterwards, the line `ram_alloc_set(aos_ram_alloc_aligned)` will make sure that from now on, any request to allocate a region of memory will make use of your implementation by using `mm_alloc_aligned`. You will find that the initial RAM capabilities provided by the kernel are quite large (typically multiple Megabytes), whereas the typical object size to be allocated is much smaller (e.g., 4 KiB for individual pages or 16 KiB for an L2CNode). Luckily, capabilities can be split into smaller chunks using the `cap_retype` invocation. You must make sure that your memory manager does not introduce unnecessary fragmentation by splitting capabilities into smaller regions first.

Have a look at the capability types and their corresponding sizes. You will notice that many of them have a fixed size (e.g., L2CNodes), while others support variable sizes (e.g., Frames). So you will need to support allocation requests of variable sizes. The second part of the allocation is the alignment: there are cases where your memory needs to have a certain alignment, e.g., a 4 KiB region must be aligned to a 1 MiB boundary. Think about how you want to support this. The allocator should return capability that has *at least* the requested size and alignment, ideally as close enough as possible – otherwise you'd be wasting memory. Make sure you discuss these kinds of aspects in your report.

### Tracking of meta-data

A side effect of splitting capabilities is that you will now require additional slots for storing the newly created capabilities in your CSpace. Also, your memory manager will likely require additional meta-data to track the state of free capabilities. For this reason, we will provide you with a *slab* and a *slot* allocator. Study the code of these two allocators in `lib/mm/slot_alloc.c` and `lib-aos/slab.c`.

*Slab allocator.* The slab allocator is a simpler version of malloc that only allocates objects of a fixed size and needs to be refilled manually (for example by using `slab_default_refill`) in case it is out of memory. Initially, you can fill it simply with a static buffer that is big enough. Be cautious not to use the same buffer to initialize multiple slab allocators. Once you have completed the next step (mapping frame capabilities), you should change that and refill it with dynamically allocated frames in case it runs out of memory. Hint: you may need slabs to do the actual refill. Be careful about possible refilling loops. You can use the function `slab_freecount` to check how many free objects there are.

*Slot allocator.* The slot allocator is responsible for allocating space that holds the

capability meta-data as required for use by the kernel. The slot allocator can also run out of space, so you need to make sure it always has enough slots left by calling the slot refill function at the right times. Like with the slabs, you should be careful about refilling the slot allocator while refilling the slot allocator.

### Freeing memory

So far we've mostly talked about the allocation of memory. However, this is just one part of the task. Another important aspect is *freeing* previously allocated memory. Your allocator should handle returning of memory to the allocator through a call to `mm_free`, and hand out this memory in future allocation requests. In other words, your allocator should not run out of memory when you allocate and free equal amounts of memory indefinitely. Think about when and how you are coalescing neighboring free regions to support larger allocation requests again. For this part you can assume that only *full* regions that have previously been allocated are freed, and you don't need to handle partial frees of a sub-range. Moreover, you can assume that the freed memory is in fact no-where else used. Thus, only memory that has previously been allocated should be freed (and maybe you have a corresponding check in place).



Extra  
Challenge

## 3.11 Support for partial frees

In this extra challenge, we're adding support for partially freeing previously allocated memory regions. For example, a process allocates a large chunk of memory, then performs some split and retype operation and then frees up part of the larger allocation again, while keeping some of the memory allocated and in use. This in turn may require you to split or shrink the previously allocated region, and insert or grow free regions. As above, you can assume that the region being freed is in fact no-where else used.



Technical  
Details

## 3.12 The ARMv8-A MMU

Next, let's take a look in detail at the ARMv8-A MMU. We'll go into a lot more detail here than you actually need for the next task, but you'll need this information later on when you implement self-paging, so we cover it here.

When discussing the design of virtual memory and paging systems in general, it's also a good idea to have a concrete example to refer to, so it's worth going into details about how it works. The ARMv8-A MMU is actually rather complex, and so we'll focus here on the details you need to know for programming it for the course.

ARM has two documents that explain the MMU in more detail than we do here, but with less density than the Architecture Reference Manual [12]: "ARMv8-A Address Translation" [15] is the older of the two but in our opinion is a lot more helpful than the guide which supercedes it, "ARMv8-A Memory Management" [16]. The

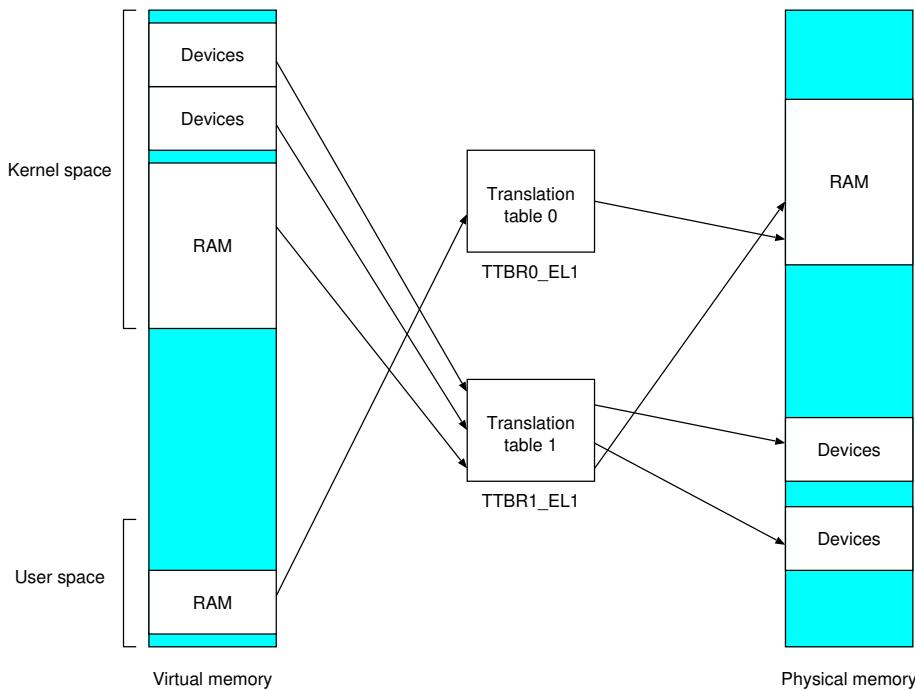


Figure 3.8: ARMv8-A address spaces in EL0/EL1

description that follows is somewhat simplified, and also assumes you've already seen a paged MMU (such as x86, for example).

### Address space layout

ARMv8-A divides the virtual address space into two parts, each of which is translated using a different page table (see Figure 3.8). The lower part has all the high bits of the address space set to 0, and is used for user-mode addresses (execution level 0, or EL0, in ARM parlance). The TTBR0\_EL1 register holds the address of the base of this page table (“TTBRO” because it's the Translation Table Base Register for region 0, and “EL1” because the register can only be accessed at execution level 1 – we don't want that an EL0 application can access this register, as it would allow unprivileged user programs to create their own unrestricted page tables and thereby access arbitrary physical memory).

The upper part of the virtual address space has all the high address bits set to 1, and is used for kernel mode (EL1) addresses. The TTBR1\_EL1 register holds the address of the base of this page table.

Virtual addresses in ARMv8-A are 64 bits in size, but at most only 48 bits (currently) are used. The sizes of the regions translated by TTBR0 (starting at address

`0x00000000'00000000` upwards) and by TTBR1 (down from `0xffffffff'ffffffff`) are set by the Translation Control Register TCR\_EL1, only accessible in the kernel.

In Barrelyfish we configure TCR\_EL1 to use a full 48-bit virtual address space, where TTBR0 translates virtual addresses `0x0000'00000000` to `0xffff'ffffffff` and TTBR1 translates `0xffff0000'00000000` to `0xfffffff'ffffffff`.

Moreover, we program TTBR1\_EL1 to map the whole of physical memory one-to-one starting at physical address zero, so that converting between virtual and physical addresses inside the CPU driver is trivially simple. On 32-bit architectures, we can't do this since the physical address space is too big, but on ARMv8-A we have the room and take advantage of it.

You might recall that on Intel architecture machines, there is only one page table, so why this extra complexity? In ARMv8-A architecture, if the MMU is on, it's on for the whole of the address space. In a Unix-like OS, every process has its own, different virtual address space, and thus its own different page table. The problem starts when a program traps and enters the kernel. If there was only one page table, either every kernel entry would require reloading the Translation Table Base Register (TTBR), or each process would have to replicate the complete kernel page table and update it each time it changed (though the latter is hopefully rare). In either case, a process switch would involve discarding all TLB entries relating to the kernel, which is a significant performance hit.

For this reason, many processors provide two TTBRs, one for the kernel, and one for user space. On a context switch, the user space TTBR is changed, but hopefully the kernel TTBR (TTBR1 in this case) is never written to after initialization and boot.

In this project, you should only need to modify the TTBR0 page table. Note, a user-space process cannot access the TTBR0 directly, as this would allow changing its own translation tables and effectively break the desired protection mechanism. To modify the page tables, a process needs to use the corresponding capability operations.

## Page sizes and Granules

Some MMUs (like those on Intel processors) have a fixed page size, which is the smallest unit of memory that can be mapped independently – it's often 4KiB.

In contrast, ARMv8-A allows 3 different “granule” sizes: 4KiB, 16KiB, and 64KiB. The granule size can be different for EL0 and EL1 mappings, and like region size is determined by fields in the TCR register. Multiple granule sizes introduce considerable flexibility into the page mapping scheme used, but also considerable complexity.

In mainline Barrelyfish so far, we always use a translation granule size of 4KiB. This should not stop you from experimenting, but we urge you to “keep it simple” at least at first.

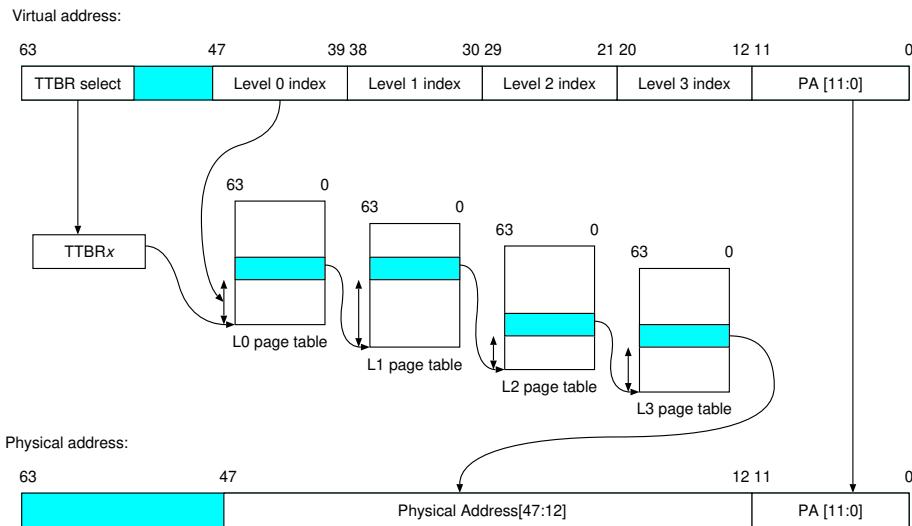


Figure 3.9: ARMv8-A 4-level page table for 4KiB granule

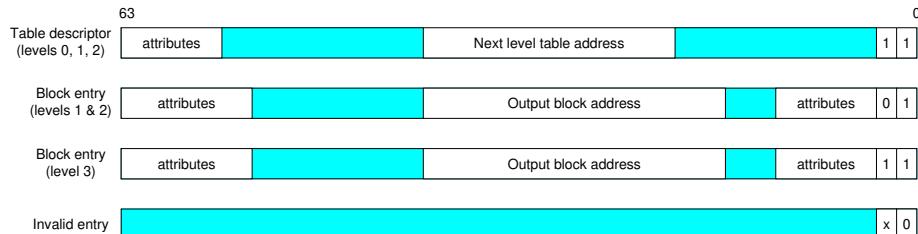


Figure 3.10: Translation table descriptor formats on ARMv8-A

Note that granule sizes are not the same as superpages. Superpage mappings are a feature of multi-level page tables where the table walk can be terminated early, resulting in a coarser granularity of translation, and associated fragmentation, in exchange for better TLB coverage and faster walks. ARMv8-A also has superpages, which are sometimes referred to in the documentation as “block translations”.

## Page table format

Assuming a granule of 4KiB in a 48-bit virtual address space, the ARMv8-A page table has 4 levels, and looks very similar to x86 (see Figure 3.9).

Note that, in contrast to x86, the first (top) level of the page table is “level 0”, and the last (bottom) level of the page table that holds individual 4KiB PTE is “level 3”.

Figure 3.10 shows the different types of page descriptor format on ARMv8-A; all are 64 bits long. The least-significant bit is 1 if and only if the descriptor is valid,

otherwise translations through this descriptor cause a page fault.

In the “normal” case, the bottom two bits are 0b11 and, at levels 0-2, contain the address of the next level of page table. Since page tables are always aligned (to 4KiB if the granule size is also 4KiB), the low bits of the page table base address are not specified, as is normal in MMUs. The Level 3 page table entry also has its low bits equal to 0b11 and holds the physical page number corresponding to the virtual address.

What about superpage mappings? On ARMv8-A, the final result of a table walk is called a “block entry” and is allowed at any level of the page table other than Level 0 (where they would be really huge pages). At levels 1 and 2, these superpage mappings are distinguished by having their two low bits equal to 0b01 rather than 0b11. Barrelyfish uses superpage mappings in the kernel to map the entire physical address space.

As with any MMU, there are plenty of other control bits and flags inside each Page Table Entry (PTE) – see the ARM reference for details. However, notably absent are bits indicating whether a page is *dirty* (i.e. the memory has been written to through this page translation) and or if it has been *accessed* (read or written). You may be familiar with these bits from the x86 MMU design, but they don’t exist on ARM. If you want to keep track of accesses to virtual pages, you have to do it yourself using traps.

We’ll provide you with functions to install and remove page mappings using the CPU driver’s code for this, but you might want to look at how it is done for curiosity’s sake. It can also help sometimes to dump page tables to figure out what might have gone wrong.



Technical  
Details

### 3.13 Constructing safe page tables

Once we’ve got an index of the address space, the next step is to maintain the actual page tables themselves.

The way it works on Barrelyfish is as follows: the memory occupied by each type of page table is referred to by a different capability type, which can be derived from a RAM type. For example, to create an L0 (top-level) page table for the ARMv8-A architecture, a user process has to acquire a RAM capability to a 4KiB naturally-aligned region of memory, and then retype it to an VNode\_AARCH64\_10 capability.

The same thing happens for other architectures, such as ARMv7-A and x86, though each level of each architecture’s page tables have their own capability types.

Once you have one of these, you can invoke this method:

```

1 static inline errval_t
2 vnode_map(struct capref vnode, struct capref src, capaddr_t slot,
3           uint64_t attr, uint64_t off, uint64_t pte_count,
4           struct capref mapping)

```

The `vnode_map` method takes three additional arguments:

1. a *vnode* capability representing the page table.
2. a *src* capability.
3. a *slot number*, indicating the entry of the page table to be updated
4. a set of *attribute* flags.
5. an *offset* into the *src* capability
6. the number of entries to map (*pte\_count*)
7. an empty slot for storing the *mapping* capability

To continue with the ARMv8-A example, in the normal case the target capability will be of type `VNode_AARCH64_13`, in order to create the leaf level of the page table. If this is the case, the correct L2 PTE will be written by the CPU driver into the requested slot in the L2 page table. The PTE's other fields will be set according to the specified flags, including access rights such as readable, writable, or executable.

The safety property is enforced by only allowing certain combinations of capability types (and sizes) to be used for `map()`. In the simplest case, the only valid target capability type for `map()` on a capability of type `VNode_AARCH64_12` is `VNode_AARCH64_13`.

To map a 4KiB page of memory, you must call `map()` on a capability of type `VNode_AARCH64_13` passing a 4KiB aligned capability of type `Frame`.

However, in Barrelyfish we allow more, in order to satisfy the completeness property. For example, you can pass a 2MiB aligned `Frame` capability to a `VNode_AARCH64_12` capability to create a block mapping.

In some cases we can even allow more exotic mappings. Sometimes, we allow a `CNode` to be mapped just like a `Frame`, but forced to be read-only, which makes debugging somewhat easier. It is an interesting question to what extent this constitutes a security risk.

On other architectures, the situation is much the same: arbitrary valid page tables can be constructed safely, as long as you hold the right capabilities with the right types.

On x86 machines, we can map the page tables themselves as if they were frames (again, the CPU driver forces them to be mapped read-only), which gives a user program access to the PTE data structures themselves. Again, this raises an interesting question: is allowing a user program to see actual physical addresses of pages a security problem?

However, it does have a practical use: an x86 PTE contains flags named `accessed` and `dirty`, which are set to 1 whenever the page is read from or written to respectively. This can be used (as in [10]) to provide valuable information to a garbage collector, or (as in [48]) to detect read and write sets for in-memory transactions. Unfortunately, ARM Memory Management Unit (MMU)s don't provide these bits.

Capability type	target type	allowed?	notes
VNode_ARM_11	VNode_ARM_12	Yes	
VNode_ARM_12	Frame (4KiB)	Yes	
VNode_ARM_12	Frame (8KiB)	Maybe	If all PTEs fit
VNode_ARM_11	Frame (1MiB)	Yes	Superpage (section) mapping
VNode_ARM_11	VNode_ARM_11	No	Invalid page table would result
VNode_ARM_12	VNode_ARM_11	Maybe	Read-only, if all PTEs fit
VNode_AARCH64_10	VNode_AARCH64_11	Yes	
VNode_AARCH64_11	VNode_AARCH64_12	Yes	
VNode_AARCH64_11	Frame (1GB)	Yes	
VNode_AARCH64_12	VNode_AARCH64_13	Yes	
VNode_AARCH64_12	Frame (2MiB)	Yes	
VNode_AARCH64_13	Frame (4KiB)	Yes	
VNode_AARCH64_13	Frame (8KiB)	Maybe	If all PTEs fit
VNode_AARCH64_13	VNode_AARCH64_1x	Maybe	Read-only, if enabled
VNode_x86_64_pml4	VNode_x86_64_pdpt	Yes	Even on ARM processors!
VNode_x86_64_ptable	L2CNode	Yes	Read-only, if enabled

Table 3.1: Examples of `map()` combinations

See Table 3.13 for more examples of what might work, and what might not, with the `map()` method. Note that, in principle, Barelfish allows you to securely build a valid 64-bit x86 page table while running on an ARMv7-A processor (and vice-versa). In practice, this requires you to have configured your Barelfish tree for both architectures. This is a powerful feature, and is a key part of Barelfish's support for heterogeneous processors which share physical memory.

Note that in the simplified version of Barelfish that we supply for the course, all this flexibility (and a lot of the associated complexity) has been removed, and we only include support for ARMv8-A page tables.



Project  
Instructions

## 3.14 Task 2: mapping frame capabilities

Once you have completed an initial version of your memory manager, it is time to write a first basic mapping function for frame capabilities. In the following milestones you will build a complete virtual address space management system, but for now it suffices that you are able to map a frame capability at a free location in your virtual address space. Similarly to the physical memory manager, the virtual memory subsystem has two high-level parts:

- Allocate and free regions of virtual address space, and

- Performing the capability operations to construct the page tables.

For this milestone, you can focus on the second part: constructing the page tables. Implement the `paging_map_fixed_attr` in `lib-aos/paging.c`. With a working version of `lib/mm`, you can now allocate VNode capabilities for different page table levels (capability types `ObjType_VNode_AARCH64_11`, `ObjType_VNode_AARCH64_12`, and `ObjType_VNode_AARCH64_13`) as well as frame capabilities and use `vnode_map()` to construct the virtual address space.

Your code basically needs to keep track of the page tables that you allocate and map. Trying to map an entry that has already been mapped will result in an error. Thus, you will need to somehow remember where your page tables are, or in other words the corresponding VNode capabilities, to support subsequent mappings in a close-by virtual address range.

For simplicity, this week you can still assume that the frame you are trying to map always fits inside a single L3 page-table and the virtual address is chosen such that it does not overlap (i.e., you do not have to modify two L3 page-tables in one mapping). Further, you might fix the first three levels of the page table (L0-L2) and limit the amount of memory that you can map for now. This will minimize the amount of meta-data you have to track for your initial mapping implementation. But, note that already in the next milestone those assumptions likely no longer hold, and you will be required to handle mappings that span multiple L3 page tables. Recall, the ARMv8 architecture supports multiple page sizes, in this milestone you are only required to implement mapping of base pages. If you are up for a challenge, you can try with superpage mappings.

However, you can't just create a new page table and map a frame. The `init` process is already using a page table that was created by the kernel—so you will need to create a mapping in the page table that `init` is currently using. For this, you will need to get hold on the corresponding capabilities. The kernel will place the root page table in a well-known location: slot 0 on the page cnode. You can refer to it by using the `cap_vroot` capability.

Note: you should have all the state of your implementation in the `struct paging_state` of which you'll find an instance called `current` setup as a global variable in the file `lib-aos/paging.c`. Also, have a look at the different comments that are marked `TODO` in `lib-aos/paging.c` to see what types of functionality you will need in later exercises.

After you have verified that you can map frame capabilities in your address space, you can now implement `slab_refill_pages` in `lib-aos/slab.c`. This will add the necessary functionality in the existing slab allocator to refill it dynamically if it ever runs out of slab space. And it will serve as a first test for your frame mapping functionality. You should have all the necessary mechanisms to implement `slab_refill_pages` by using a combination of the two functions `frame_alloc`, `paging_map_fixed_attr` (for now use a manually chosen VA offset), and `slab_grow`.

Once completed, you should go back and verify that your code works by writing a

test that checks if you correctly make use of the provided slot and slab allocators in your `libmm` implementation. For the slot allocator, you can check that your code can handle more than 256 allocations. For the slab allocator you can similarly try to exhaust the initial static buffer space to force a refill (make sure the slab allocator will use your `slab_refill_pages` function). Be aware, it is easy to step on your own feet, e.g., you are refilling your slot allocator requiring the allocation of a slab, which in turn triggers slab refilling that uses another slot.



### Extra Challenge

## 3.15 Superpage Mappings

Recall, ARMv8 supports superpage mappings (or huge pages in Linux terminology). Instead of installing the mapping in the level-3 page table, you will install a mapping in the level-2 page table directly. In this extra challenge, you will need to handle mappings of two page-sizes: 4 KiB and 2 MiB. Moreover, you may need to adapt the memory allocator to support allocation requests of naturally aligned 2 MiB frames. Demonstrate that you can allocate and map 2 MiB frames using superpage mappings. Think about how your paging code will support selecting superpage mappings. This entails requesting superpage mappings explicitly, and a policy that will decide when to use superpage mappings when not requested explicitly. Note, the kernel currently doesn't have support for superpage mappings, so you will need to add it! Be warned, getting superpages is not trivial, as there are many cases to consider. Make sure that your paging code works with base pages first!



### Commentary

## 3.16 How we test

As with most of the following milestones, the code you will write will be tested by us using a standard interface. In the case of this milestone, our tests operate on the following function calls:

```

1 errval_t mm_alloc_aligned(struct mm *mm, size_t size, size_t alignment,
2                               struct capref *retcap);
3 errval_t mm_alloc(struct mm *mm, size_t size, struct capref *retcap);
4 errval_t mm_free(struct mm *mm, struct capref cap);
5
6 errval_t paging_map_fixed_attr(struct paging_state *st, lvaddr_t vaddr,
7                               struct capref frame, size_t bytes, int flags);

```

Make sure to not change any of these function calls, otherwise our tests will fail (and you will lose points!).

The handout contains function calls to the grading library (`grading_setup_bsp_init()`, `grading_test_mm`, `grading_test_early()` and `grading_test_late()`) in `init`. Make sure to initialize the memory management library before the `grading_test_mm` and `grading_test_early` functions are called. For further details on when to call

the grading functions, have a look in `include/grading.h`. In your handout, the grading library is a stub, which will be replaced by our testing code once we evaluate your submission.

## 3.17 How to do well

Since this is the first assignment with real design and coding involved, it's worth talking about what will get you lots of credit in the course.

It's important to think like an OS designer: what can go wrong? What are the corner cases? What needs to be fast (particularly when the system gets large)? What functionality might be missing (such as freeing up resources), and how can it be implemented?

In this case, you should be able to demonstrate the following:

- init allocating memory for the various capability types.
- Tracking of free capability slots and being able to re-allocate them again.
- Refilling of the slab and slot allocators (which can be shown by allocating a large number of capabilities).
- Allocation of a frame capability that gets mapped at a fixed location.

Think about what data structure you are going to use for the memory manager, and be prepared to describe what the structure members all do. For example, you might use a linked-list, heap, stack or tree for this. While some form of linked list might be easiest for storing free capability slots, what are the performance implications?

Writing tests that establish that `mm_alloc()` and `mm_free()` work as advertised will get you points (recall, that you will need to demonstrate the functionality somehow), but also give you confidence that the code is correct. Similarly, mapping a single 4KiB frame is important, but just as important is mapping many more of these.

Finally, if you skip forward to Chapter 14, you'll see that at the end of the course you have to deliver a substantial report on what you've done. Start now! Write down your experiences, decisions, wrong turnings, performance measurements, test cases, etc. as you go along, and you'll make finishing the final report much easier.

## 3.18 Milestone 1 Summary

Milestone tasks:

1. Implement `libmm` to manage physical memory.
2. Demonstrate that your implementation can allocate and free RAM capabilities of various sizes (multiples of base page size).



Commentary



Project  
Instructions

3. Demonstrate the ability to refill the slab and slot allocators.
4. Show that you can map Frame capabilities of different sizes in your address space within the same leaf page table.

Extra challenges:

1. Support partial frees.
2. Support superpage mappings (2 MiB large pages)
3. Find and fix any bugs you can in our code (ongoing).

Submission:

You should submit your code as specified on the course website before the specified deadline, and be prepared to demonstrate your code in the marking session.

Assessment:

1. We'll test the allocation and reclamation of RAM capabilities in `init` for various capability types and allocation patterns.
2. You should be able to explain the design of your memory manager as well as any of the provided code you have used (i.e., slab and slot allocators).
3. You need to show you can map and write memory in `init` and be able to explain the code you have written to do that.

## **Part II**

# **Group work: Building the core of the OS**



The previous assignment completes the purely individual part of the course. From now on, you'll be working in a team of 3 or 4 people – first in building common components of your operating system on top of the foundation you have, and then in Part III building individual components that work together in a complete system.

If you haven't done so already, now is the time to form your team.

You should also pick a foundation (an implementation of the previous milestone) on which to build the rest of your operating system. You can pick one team member's version (if the rest of you trust them), or combine ideas from everyone's individual milestone implementations, or indeed (if you have plenty of time and enthusiasm) redo the whole thing from scratch.



## Chapter 4

# Processes, threads, and dispatch

In the previous milestone, you built a simple allocator for physical memory running the first process to start, `init`. The `init` process is a special case in that it is the only process started by the OS kernel (in Barrelyfish the kernel is known as the *CPU driver*, for reasons that will become clear later).

### 4.1 Process models

Operating systems create processes in different ways, though the basic set of steps is common to most. In almost all cases, the first process is special, and is created in a fairly ad-hoc manner by the kernel.

In a Unix-like OS like Linux, all processes after `init` are created in the kernel by forking an existing process. Unix then allows an existing process to be replaced by a new program (generally loaded from the file system) using the `exec` call. This combination of `fork()` and `exec()` is arguably the defining feature of Unix.

In contrast, Windows generally starts a process from scratch, given a binary file to run. Like Unix, however, this is done in the kernel.

In Barrelyfish, a new process is created by an existing process which allocates space for all the process data structures (including the process control block, the text segment holding the program code, the stack, data, and bss segments) using the capability facilities we saw last week.

In this milestone, the goal is to start another process in this way. At bootstrap time, the initial process `init` is created and started by the CPU driver. Your task now is to create a subsequent process from `init`.



Commentary

Exactly what a process looks like and how it is created depends on two different parts of the OS design.

The first is the *kernel execution context*: what happens in terms of stack, registers, etc. when the processor enters kernel mode as a result of a system call or trap, and how it swaps context between processes.

The second is the *user threading model*: how are user-visible threads implemented given the process contexts provided by the OS.



## Background

There is typically a vast difference in perspective and skill between programmers who have written an OS and those who haven't: In regular programming, threads, the stack, memory management, etc. are all provided as if by magic by the compiler, language runtime, and OS. In the kernel, the programmer has to do all of this herself.

Some key design choices for an OS kernel are:

- Is there support for more than one execution context in the kernel?
- Where is the *stack* used for executing in the kernel?
- Can code in the kernel *block*? If so, how?
- What happens if an interrupt is raised while in kernel mode?

The set of answers to this questions is often called the *Kernel thread model*. There are two common alternatives:

1. A per-thread kernel stack: every process or thread has a matching kernel stack.
2. A single kernel stack: there is only a single kernel stack (per core, in a multi-processor OS), used by every process running over the kernel (on the core).

### 4.2.1 Per-thread kernel stacks

In the first alternative, every user thread or process has its own stack in the kernel as well as in user space, and the thread's kernel-mode state is implicitly stored on its kernel stack. If a thread executing in kernel mode needs to block (for example, because it needs to wait for I/O to complete before the current system call can return to user mode), the kernel switches stacks at that point to a kernel stack for a different, previously blocked thread and resumes that. Later on, when the former thread can be unblocked, it's just a matter of switching kernel stacks back again.

This makes it easy to preempt code executing in the kernel, which is important when kernels are very large (as in Linux) and/or execute long-running and complex operations inside the kernel which often have to wait on I/O (as in Linux).

It also means that there is very little difference between how programmers write kernel-mode code and user-mode code, except for the small section responsible for actually performing the stack switch – system call code might resemble the following pseudo-code:

```

1 example(arg1, arg2) {
2     P1(arg1, arg2);
3     if (need_to_block) {
4         thread_block();
5         P2(arg2);
6     } else {
7         P3();
8     }
9     /* return to user */
10    return SUCCESS;
11 }
```

Interrupts are also often enabled in such kernels, and indeed kernel code can sometimes take page faults as well. Exceptions in general are handled the same way as in user space. This lead to the familiar Unix split of interrupt service routines into “bottom half” and “top half” – the bottom half is the first-level exception handler and runs when the interrupt is raised, but for this reason cannot take out locks or access most other kernel resources. Instead, it generally schedules another piece of code to run at a more convenient time (usually once a process is running again, just before it exits the kernel) to handle the rest of the interrupt.

For this reason, per-thread kernel stacks also make sense when the kernel is not merely large, but also written by a large number of programmers, many of whom have had little experience with alternative forms of OS design or kernel programming (as in Linux).

The disadvantages of this model include increased size of kernel: the code is larger, partly because it needs to support multiple threads, but also that it encourages more complexity to move into the kernel (since it's so easy). The data used by the kernel is also larger, because of all those kernel stacks.

This is not just a problem with running out of DRAM: the big problem these days is the increased cache footprint of the OS, which slows down any system which spends significant time in systems-level code.

Nevertheless, many mainstream and non-mainstream operating systems adopt this model, including Linux (and most other varieties of Unix), L4Ka::Pistachio, etc.

### 4.2.2 Single kernel stack designs

On a uni-processor machine, the alternative to having a different kernel stack for every process (or thread) is to have a single stack which is used for everything in kernel mode. Every time the kernel is entered (via a system call, interrupt, page fault, etc.), this stack is used for executing code.

The key challenge with this design is how to handle system calls that block. There are two basic approaches: using *continuations*, and the so-called *stateless kernel* approach.

### Continuations

Continuations are a very old idea in Computer Science, and a very powerful one. They are quite familiar to people used to functional programming languages (in particular, Scheme). They can be thought of as a generalization of pretty much all imperative control flow constructs (call-and-return, coroutines, etc.), and there is a long tradition of compilers generating intermediate code that consists almost entirely of continuations (CPS or *continuation-passing style*).

A continuation for our purposes is a first-class object which encapsulates the state of an execution. Given a continuation, you can pass it around just like any other data, but you can also “call” it, which causes execution to resume at the point in the program referred to by the continuation.

Strictly speaking, the C language doesn’t have continuations, but they can be approximated in various ways (sometimes with the aid of some assembly language). For example, a `jmp_buf` (as used with `setjmp()`) is a limited form of continuation: it can only be used once, and you have to call it in a funny way by calling `longjmp()`.

If the program (the kernel, in this case) is written carefully, a *closure* (in other words, a function pointer and a set of optional arguments) can be used for this purpose. When using continuations to implement blocking calls in a single-stack kernel [38], the code looks a bit like this:

```

1 syscall_example(arg1, arg2)
2 {
3     P1(arg1, arg2);
4     if (need_to_block) {
5         save_context_in_PCB;
6         process_block(syscall_example_continue);
7         panic("process_block_returned");
8     } else {
9         P3();
10    }
11    process_syscall_return(SUCCESS);
12 }
13 syscall_example_continue()
14 {
15     recover_context_from_PCB;
16     P2(recovered arg2);
17     process_syscall_return(SUCCESS);
18 }
```

What’s happening here is that when the code realizes it needs to block (line 4), it saves the current process’s context in its Process Control Block, and then calls

the scheduler to figure out which process to run instead (line 6), passing as an argument a pointer to a function `syscall_example_continue`). This function is where the process should start executing next time it is dispatched by the scheduler – it doesn't ever expect to continue exactly where it left off (line 7).

To see the continuation here requires stepping back a bit and squinting, but it consists of the `syscall_example_continue` function, *plus* the state saved in the PCB.

The reason it looks somewhat convoluted (why the panic? Why can't the continuation simply follow the call to `process_block?`), is that the stack is reused – when `syscall_example_continue` is called, all the previous stack frames have been thrown away (and the space they occupied reused many times in the intervening time).

A continuation-based kernel has a much lower cache and memory footprint, due to there only being one stack. However, it's complex to program: the flow of execution of a single process doesn't correspond to the flow in the code you read in a listing. The sequence of continuation calls is sometimes known as "stack ripping" (since one is basically giving up on the convenience of a stack!), and we'll see this pattern again when we look at inter-process communication in user space.

Another disadvantage of continuations is that you need to be very conservative about what state you save in the continuation, since you don't know in advance what is needed. If not all the state needs to be preserved (for example, if the scheduler decides you can unblock anyway), you've done a lot of unnecessary work.

In smaller kernels, though, this design does make sense. It's used in, among others, Mach [38] (one of first generation of microkernels) and NICTA::Pistachio L4 [96].

Interrupts can be handled in two ways. You can use the multi-stack kernel approach, with a bottom half and top half handlers, or simply disable all interrupts while in the kernel, and so handle each interrupt sequentially.

### Stateless kernels

In a stateless kernel [47], the problem of blocking doesn't arise: system calls never block. If a system call cannot complete within a "reasonable" time, it fails, and it is the job of the calling process to try again when resources become available.

This doesn't mean that the system call had no effect. It's possible that it returns with a result code which implies "you asked for A,B, and C; I managed A, so you need to try again with B and C". This makes a stateless kernel a bit like one based on continuations, except that the continuations are pushed back up to user space. A system call which fails to complete instead returns a continuation to the user process, which they can then pass back to the kernel later on to continue the operation.

Because either a system call always succeeds or returns a continuation fairly quickly, the kernel stack contents can be completely discarded after each system call or exception – the kernel always starts afresh. This is why the kernel is described

as “stateless”: any mid-call state is always handed back to the calling process for safekeeping rather than storing in the kernel.

Preemption inside a stateless kernel is difficult to handle: either the currently-executing system call has to roll back to some restart point and create a continuation, or the kernel must be designed very carefully. Typically stateless kernels do not take page faults (i.e., all memory the kernel touches is pinned and mapped into the kernel’s virtual address space).

seL4 [66] is a good example of a stateless kernel with limited preemption. The seL4 designers wanted the kernel to be preemptible because some kernel operations could run for a long time, and turning off interrupts for a long time could harm real-time performance (for example, in a mobile phone).

The canonical example of a potentially long-running operation in seL4, and indeed Barreelfish, is the deletion of a capability (an operation you saw in Chapter 3). This requires deleting all derived children of the capability, and this can take some time. In order to guarantee forward progress, seL4 avoids returning until *some* forward progress has been made in deleting the capability, even if it didn’t manage to complete the job. It does this using clever rearrangements of the capability tree so that something can always be deleted.

Another example is the zeroing of frames of memory (i.e., when a RAM capability is retyped into a frame), to prevent a process from seeing old data from a previous user of the page).

Stateless kernels share the small cache and memory footprint of continuation-based kernels, but may be harder to develop due to the complexity introduced by preemption. Any preemptible system call can only be preempted at certain well-defined “preemption points”, at which the system invariants are guaranteed to hold, and between which forward progress is guaranteed to occur.

Another issue that might arise is if the processor architecture “manages” the kernel stack in ways that work against the stateless approach. This has to be worked around in the kernel entry/exit code, and is the kind of hardware feature that can really annoy system software designers (you will encounter other such annoying pieces of hardware design throughout the course).

Despite this, if one can guarantee by design that the handlers for system calls, interrupts, and other processor exceptions are all:

- atomic,
- non-blocking,
- execute in a short and *bounded* time, and
- will never cause a page fault

– then a stateless single-stack (per core) kernel is the simplest and most elegant model. Unfortunately, none of the above criteria apply in the case of Linux.

What about multiple cores? We've seen that a single kernel stack can make sense on a single-core machine, but what about multiple processors? After all, the number of machines these days with only a single processor core is vanishingly small.

A single stack can't be shared between multiple cores executing all at the same time. The obvious solution is to have one dedicated kernel stack per physical core, and then deal with the complexity of migrating a process from one core to another if required.

## 4.3 Barrelfish kernel architecture

Barrelfish adopts a radically simplified approach to kernel architecture, by adopting a “purist” stateless kernel approach (in common with its predecessors Nemesis [72], K42 [67], and the Aegis Exokernel [44]).

The Barrelfish kernel (known as the “CPU driver”, since that name reflects the simplicity) has the following properties:

- The CPU driver runs on a single core, and has a single (kernel) stack.
- No system calls block: each one is run to completion.
- All interrupts are disabled when in the CPU driver. Moreover, the CPU driver never takes a page fault or any other trap while executing its own code.

You can think of the CPU driver as a program which handles one processor exception (page fault, system call, interrupt, etc.) in turn until it has nothing more to do, and then runs a user process until a new exception turns up.

The problem of long-running operations was solved somewhat differently. Some (such as zeroing pages) could be pushed to a trusted user-space process. In other cases (such as capability deletion), the operation is explicitly broken up into atomic steps in advance, rather than allowing it to be preempted at specific points.

This can result in more overhead (lots of system calls), but the motivation was (a) this is simpler, (b) such operations are latency-sensitive, and (c) in the case of capability operations (which are most of the long-running operations the kernel has to perform) we will see in Chapter 7 that in Barrelfish these *have* to be split-phase distributed operations anyway, which are handled by a mediating process (called the Monitor).

## 4.4 Creating a process

You'll be building process management functionality above the slightly simplified Barrelfish CPU driver we gave you in Chapter 3.

To create a process, the following high level steps are needed:

- Finding the appropriate ELF image for the program.



Technical  
Details



Project  
Instructions

- Creating an initial CSpace for the new process, so that it has all the capabilities it needs when it starts to run.
- Creating an initial VSpace layout, so that when the process starts running its virtual address space is sufficiently complete to execute the code and access its data.
- Loading the ELF image containing the program for the new process into both parent and child address spaces.
- Adding a *dispatcher* for the new process, and telling the CPU driver about it so that the new process can be scheduled and dispatched.

Much of the work of creating a new process is memory mapping operations and keeping track of your own process (in the first case, `init` and your child process' virtual address spaces). To do so, you will extend the memory management functions that you have started to use in the previous milestone.

Note, that full self-paging functionality will be part of a later milestone.

This is one of the longer milestones. As you can see, creating a process can be a complex business. We strongly suggest starting early!



#### Background

## 4.5 Fork or spawn?

You'll notice that the steps outlined above rather laboriously construct a new process and its program from scratch. This operation is generally known as "spawning" a process, in contrast to the famous Unix "fork" operation which you're probably familiar with.

In an interesting way, `fork()` really is the essence of Unix. If you want to implement Unix, almost everything is based around `fork()`. If you base your new design of OS around `fork()`, you *will* write Unix, trust me. Non-Unix operating systems, notably Windows, don't have `fork()` – they all use the spawn model.

It's pretty common for a Unix `fork()` to be immediately followed, in the child, by an `exec()` to replace the child process image with a new program – so much so that a number of attempts were made in the past to optimize this. After all, creating a complete copy of a running process so that one can immediately throw it away and load a new program seemed rather inefficient, hence the extensive use of copy-on-write to create the illusion that one is doing something terribly inefficient when it's not that slow in reality (though it's not that fast either).

`fork()` was, originally, a quick hack to get processes running on a computer without paging (only swapping) [20]. While many textbooks have eulogized `fork()` as a great piece of elegant design, it is riddled with serious problems and inefficiencies on a modern, multicore machine with gigabytes of DRAM, running a mix of complex, multi-threaded workloads. When it was written, it was a simple bit of im-

lementation by one of the world's great computer scientists and programmers, but today it's arguably a very bad idea. But that's an interesting debate for another time.

One interesting feature of Barrelyfish, in contrast to other spawn-based OS designs is that creating a child process is done by the parent, not by the kernel. This is possible (as you may have guessed by now) because of the capability system, although Nemesis also used this model.

## 4.6 Extend the paging code

In the previous milestone you mapped a frame (i.e., a physical page) into your own virtual address space at a virtual address provided by the caller.

In this milestone, you'll need to extend this code in three ways:

1. Support of the *allocation* of free regions of virtual address space.
2. The ability to map large frames at a free, to-be-allocated region of virtual address space, that includes handling the case where the frame spans multiple L3 page tables.
3. The functionality to perform those allocations and mappings in a different “domain” (which, for the moment, is the Barrelyfish equivalent of a process)

In short, you'll need to manage two address spaces: the one of the spawning process and the one of the process to be spawned, and the ability to map a large (e.g., 5 MiB) frame into both address spaces.

When doing this, you must not change the function signatures as provided in `paging.h`. There is no need to, and in any case they are used in various other places in the code.

You might want to have a look at the function `spawn_load_with_bootinfo` from upstream Barrelyfish in the file `lib/spawndomain/spawn.c`<sup>1</sup>.

As well as simply creating mappings, you will also need to implement code to keep track of your virtual address space. In particular, you need to keep track of which virtual addresses are already allocated and which ones are free. For this purpose, you need to implement the function `paging_alloc` in `paging.c` which returns a free virtual address region of the requested size. When doing this, note that you can (and probably should) add additional needed state to `struct paging_state`. Note, that you should track *all* mappings in an address space, i.e., also the ones created using a user-provided virtual address (e.g., `paging_map_fixed_attr`).

Your virtual address allocation should start at the address `start_vaddr` provided to the `paging_init_state` function.



Project  
Instructions

---

<sup>1</sup><https://github.com/BarrelyfishOS/barrelyfish/blob/master/lib/spawndomain/spawn.c>

When spawning a new domain you'll need to map a frame into *foreign* address space. There are two aspects that are important here: first, the mapping must be installed in the page tables of the new domain, and second, the corresponding capabilities should be in the CSpace of the new domain. We'll discuss setting up the CSpace later in this milestone. Thus, you will also need to implement the function `paging_init_state_foreign`, which initializes the same state, but in a child process rather than the calling process.

Once you have written all three functions, you can then implement the function `paging_map_frame_attr`, that finds a free virtual address and maps a frame at that address.

Note, that we'll revisit the paging code in a later milestone where we'll implement self-paging functionality. For now, focus on the functionality required to spawn a new domain: mapping the ELF binary and the buffer where it's loaded at into the own address space, and mapping the buffer with the loaded ELF binary into the foreign address.



*Extra Challenge*

## 4.7 Unmapping

To spawn a new domain, all you need is the ability to map memory frames in the address space of the spawning process. While this is sufficient to handle a few spawns, you will eventually run out of address space. Thus, cleaning up after spawning a new domain, i.e., removing unneeded mappings, is the right thing to do.

For this extra challenge, implement functionality to unmap a previously mapped frame. This entails to modify the page table and clear the mapping, and free up the region of allocated address space.



*Project Instructions*

## 4.8 Find and map the ELF binary

We can now move on to loading a program image in an address space.

The code you write for creating a process or domain should be put in the library `lib/spawn/`. Specifically, you should provide an implementation for the function `stub_spawn_load_by_name` in `spawn.c`.

The init process will use functionality from `lib/spawn` so you should add the library `spawn` to the Hakefile for the `init` program.

Make sure to include a binary for the process you want to create in your boot image. A good first candidate is the `hello` program that should be already in your source tree. To include it in the Toradex board boot image, modify the platform Hakefile `platforms/Hakefile` to add it to the `modules_common` list, but replacing this line:

```
1 modules_common = [ "/sbin/" ++ f | f <- [ "init" ] ]
```

– with:

```
1 modules_common = [ "/sbin/" ++ f | f <- [ "init", "hello" ] ]
```

Also, add the following line to the `platforms/arm/menu.1st.armv8_imx8x` file in your build directory:

```
1 module /armv8/sbin/hello
```

If you want to add a program to the default set of programs that get added to the boot image, you can instead add this line to the file `hake/menu.1st.armv8_imx8x` in the source directory.

In Barreelfish, we adhere to the “multiboot” specification for loading the modules of our operating system. Multiboot is essentially a way to combine a collection of “modules” (think of them as files) into a single boot image, so that code (in this case, `init`) can later extract them individually. The program to run in your newly-created process will, at this stage, be stored in a multiboot module. Later in the course, you might be able to load programs from the file system instead. All the information about a multiboot image (and, in particular, where all the modules are) is represented in memory using a `struct bootinfo` datastructure.

You can use the multiboot functions defined in `lib/spawn/multiboot.c`. To locate a module by name use the function `multiboot_find_module`. This library maps the multiboot modules in frames, so it will not work correctly unless you have implemented the previous step.

When the CPU driver creates the `init` process, the CPU driver maps the `struct bootinfo` record into your `init` process’ virtual address space. The (virtual) address of this record is then passed to `init` as the first command line argument, so you can get a pointer to the `bootinfo` structure as follows:

```
1 bi = (struct bootinfo*)strtol(argv[1], NULL, 10);
```

Once you have obtained the `struct mem_region` for your module from the multiboot image you can use the `mrmmod_slot` field to create a `struct capref` to the binary’s frame.

```
1 struct capref child_frame = {
2     .cnode = cnode_module,
3     .slot = module->mrmmod_slot,
4 }
```

At the end of this step you should be able to verify that the first four bytes of your mapped binary contain the bytes `0x7f 'E' 'L' 'F'`. This is the ELF “Magic Number” – all ELF files start with this sequence.



## 4.9 Setup initial C- and V-Space

We now need to create a CSpace (i.e., the list of capabilities) and a VSpace (a virtual address space, in other words a page table) for the new domain.

The CPU driver expects the CSpace to be represented as a two level tree, so you will have to create both a Level1 CNode and a couple of Level2 CNodes that are linked in the L1 CNode for your child process – this should give you enough capability slots for the time being.

### Creating CNodes

To create the child's CNodes you can use the following functions:

```
1 errval_t cnode_create_l1(struct capref *ret_dest, struct cnoderef *cnoderef);
```

This function creates a new Level1 CNode and fills in a capability that references this CNode in `ret_dest`.

```
1 errval_t cnode_create_foreign_l2(struct capref dest_l1, cslot_t dest_slot,
2                                 struct cnoderef *cnoderef);
```

This function creates a Level2 CNode. In comparison to the `cnode_create_l1` function that you've encountered in the previous milestone, this version allows you to specify the L1 CNode you want to use (for example the one you've already created for your new process using `cnode_create_l1`) – that's why it's a "foreign" CNode

You can create a `struct capref cap` that references a slot in the Level2 CNode by setting `cap.cnode = cnoderef`, where `cnoderef` is returned from the L1/L2 create function.

### Creating the CSpace

Next, we need to populate the new process' CSpace with useful capabilities.

The newly created process must be able to find and use some of the capabilities that are created during process creation. For example, the child process must be able to find the L0 pagetable capability so that it can map memory into its own address space.

To do so, we create the CSpace with *conventions* about how it is laid out, and which capabilities are where. You can use the constants in `include/barrelfish_kpi/init.h` and `include-aos/cspace.h` to determine the location of the fixed capabilities.

The format looks as follows; note that the Barreelfish CPU driver refers to a schedulable unit as *dispatcher* – this will represent your process to the scheduler. Your process will expect the following Level2 CNodes:

1. ROOTCN\_SLOT\_TASKCN: Contains information about the process itself.
  - TASKCN\_SLOT\_SELFEP: Endpoint to itself. You can get this capability by retyping the dispatcher capability to `ObjType_EndPointLMP`. Endpoints are used for inter-dispatcher communication, and you'll use them in the next milestone. Don't worry about that for now, though, other than creating SELFEP as described here.
  - TASKCN\_SLOT\_DISPATCHER: Contains the dispatcher capability. The dispatcher is the equivalent of a "Process Control Block" in other operating systems. You can create one using `dispatcher_create(...)`;
  - TASKCM\_SLOT\_ROOTCN: Contains a capability for the root (L1) CNode.
  - TASKCN\_SLOT\_DISPFRAME: A capability to the dispatcher frame, used to communicate between a process and the CPU driver.
  - TASKCN\_SLOT\_ARGSPAGE: A page containing a list of command line arguments.
2. ROOTCN\_SLOT\_ALLOC\_0: Empty L2 Node that contains space for the child's initial slot allocator. When the new process starts creating and retyping capabilities, they will be stored in here.
3. ROOTCN\_SLOT\_ALLOC\_1: A second L2 Node for the slot allocator to use.
4. ROOTCN\_SLOT\_ALLOC\_2: A third L2 Node for the slot allocator to use.
5. ROOTCN\_SLOT\_BASE\_PAGE\_CN: Each slot holds a `BASE_PAGE_SIZEd` RAM capability.
6. ROOTCN\_SLOT\_PAGECN:
  - Slot 0: Contains a capability for the process' ARMv8-A top-level pagetable.
  - Other slots: Used to store ARMv8-A lower-level pagetables and mapping capabilities.

### The initial VSpace

Having created the initial capability set for the new domain, you now need to construct its virtual address space. This means creating hardware page tables, and populating them with mappings to physical memory.

To create a new Level0 Pagetable, you can pass `ObjType_VNode_AARCH64_10` to the following function:

```
errval_t vnode_create(struct capref dest, enum objtype type)
```

The `dest` argument should be `struct capref` referring to the first slot in the page cnode (slot `ROOTCN_SLOT_PAGECN` of the root cnode).

Now you can use the same functions to create mappings as you have used in Milestone 1 (`vnode_map` and friends). While performing the mappings, you may decide to put the created VNode and mapping capabilities into specific locations of the child's CSpace. Document your choice in the report.

Keep in mind that you can only invoke a capability that is in your own CSpace. To invoke a capability that resides in the child's CSpace, you must first copy it in the parents CSpace using:

```
errval_t cap_copy(struct capref dest, struct capref src)
```

In the end, the child domain should posses all capabilities of its VSpace in its own CSpace – otherwise the child domain would not be able to map new memory into its address space (see [chapter 6](#)). Likewise, the parent does no longer need to hold on those capabilities.



*Extra  
Challenge*

## 4.10 Passing Address Space Information

You will create certain mappings in the child's address space. However, the child process may not necessarily be aware of these mappings when it starts executing. A simple way to avoid conflicts is to prevent mappings in a certain region of the address space, or to pass the paging state to the child, which then uses this information to populate its own address space.

In this extra challenge, implement a way to pass the paging state to the child.



*Project  
Instructions*

## 4.11 Parsing the ELF

You need to look into the binary file extracted from the multiboot image to figure out information like where it needs to be located, which memory segments need to be created, etc.

You can use the `elf` library in the source tree to parse the binary. The function `elf_load` takes a pointer to the (mapped) binary and calls the callback function `allocate_func` for each ELF section.

On Ubuntu 20.04 LTS, you can use the tool `aarch64-linux-gnu-readelf` to parse the ELF and display various information. This may be helpful to compare and everify the information you've read from the `elf` library.

For `em_machine` you should use the constant `EM_AARCH64`. `base` and `size` are the address and size of the mapped binary. `retentry` will be filled with the entry point of your child process.

The callback function should allocate memory, then make the allocated memory available in the virtual address space of the process you are creating. The region of the child is described by the parameters `base`, `size` and `flags`. Note, you will

need to map the region at the fixed address base in the child domain. `flags` is a bitmask describing the rights of the child process to that region, see the definition of `PF_W` and friends in `include/elf/elf.h`. To allow the ELF library to actually write in that section, you must map the same memory into your current address space and return a pointer to that region in `ret`. The library will then copy the segment from the ELF binary into the allocated memory. Use the `state` pointer to keep track of the mapped regions, so you can unmap them later.

```

1 errval_t elf_load(uint16_t em_machine, elf_allocator_fn allocate_func,
2                   void *state, lvaddr_t base,
3                   size_t size, genvaddr_t *retentry);
4
5 typedef errval_t (*elf_allocator_fn)(void *state, genvaddr_t base,
6                                     size_t size, uint32_t flags, void **ret);

```

You will also need to find the `.got` (global offset table) section and use the address to initialize the offset registers. The GOT stores the absolute addresses of all global variables and is located at a fixed offset from the code, so it can be accessed by code generated by the compiler which uses fixed offsets from the program counter. With this table another layer of indirection is added to enable the use of shared libraries (which reside at different addresses for different processes). To locate the section, pass `.got` as section name to the function `elf64_find_section_header_name`. The returned struct contains a member `sh_addr`, which points to the GOT base address in the child's vspace. You will have to use this value later when you set up the dispatcher.

```

1 struct Elf64_Shdr* elf64_find_section_header_name(genvaddr_t elf_base,
2                                                 size_t elf_bytes, const char * section_name);

```

We're almost at the point where we can start a user process. However, this raises some important questions: what actually runs in user space? How does the OS start it, not merely when the process starts, but any time it decides to run a thread of the process on the core? Indeed, what exactly *is* a thread, as opposed to a process?

This requires a bit more background.

## 4.12 User Thread Models

We'll assume you're already familiar with programming with threads, but you may (or may not) have realized that threads can be thought of in different ways.

For one thing, threads are a **programming language abstraction** to allow the programmer to deal with activities that might be concurrent. For example, I/O requests can often take some time to complete. One way to save time is to start lots of I/O requests (sometimes thousands, for example in a web server) and give each one a thread which blocks until that thread's request has completed. In a scenario like



*Background*

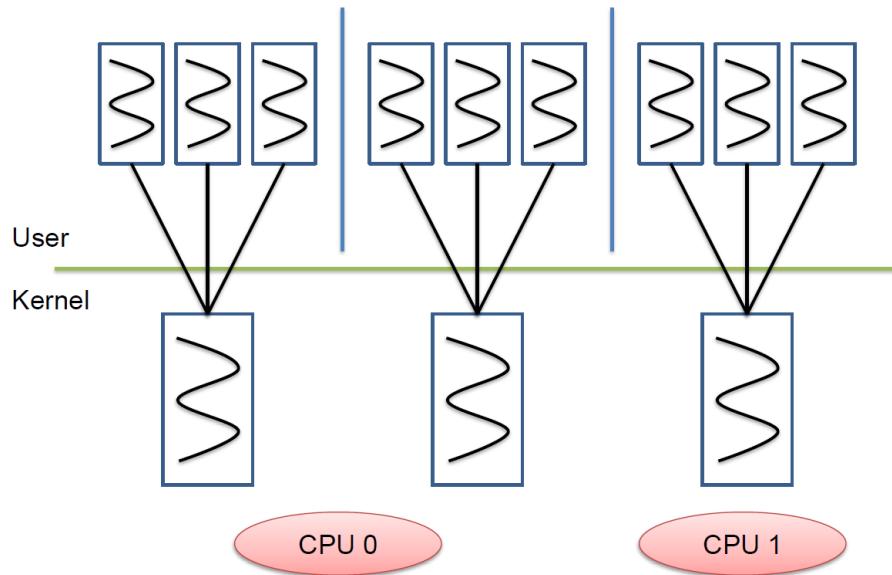


Figure 4.1: Many-to-one user-level threads

this, threads have to be lightweight: they should ideally be cheap to create, destroy, and switch between.

On the other hand, threads are also a **kernel abstraction** allowing a program to take advantage of multiple processors which can execute code simultaneously: they are effectively virtual CPUs (you'll see them described as such in many OS textbooks). As we saw earlier with kernel architectures, however, this kind of thread is typically quite heavyweight and takes up a lot of resources (memory and cache footprint, context switching by entering the kernel, etc.). Performance engineers will tell you that, when used in this way, there's no point in having more threads than you have physical CPUs in your system.

We use threads for both these purposes without usually distinguishing them, which is unfortunate, because a threading *implementation* which is designed for one case is typically not ideal in the other.

To make things more complex, threads need to communicate, and inter-process and inter-thread communication overhead is critical to the performance of many applications.

Traditionally, there are three approaches to implementing threads in an OS. In no particular order:

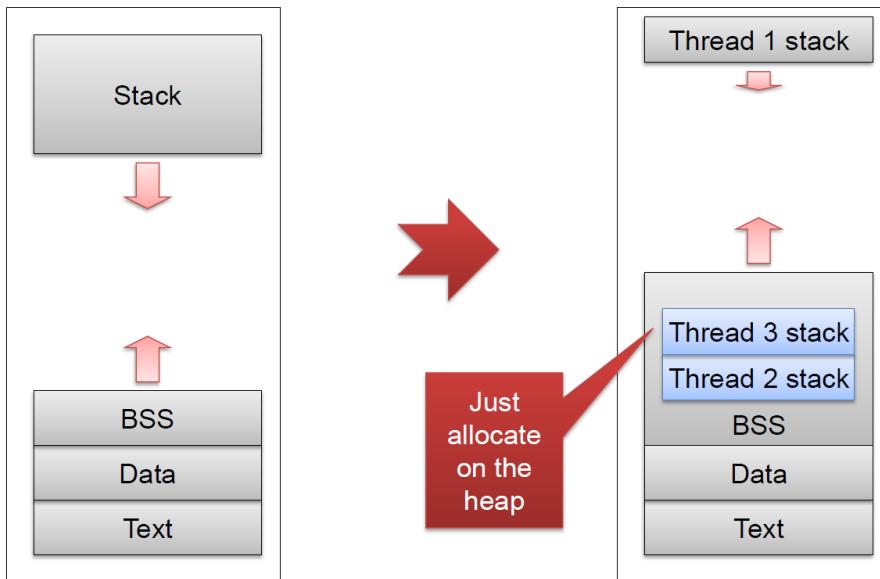


Figure 4.2: Memory layout for many-to-one threads

#### 4.12.1 Many-to-One Threads

Early thread libraries over Unix (single threaded) processes used the model shown in Figure 4.1, including the “Green Threads” package used by the very first Java virtual machine implementations – in early versions of Unix this was the only option available for multi-threaded shared-memory programming. The GNU Portable Threads library is another example of this, and it’s often a standard student exercise to try to implement many-to-one threads over Unix. It shouldn’t take a good student more than a day or so to get things mostly working...

This model of threads is sometimes called “pure user-level threads”, since no kernel support is required – the kernel is typically unaware that the process it is scheduling is actually multi-threaded at all. In some OSes (such as Solaris), such threads are also (confusingly) referred to as “Lightweight Processes”.

Pure user-level threads are laid out in memory as shown in Figure 4.2. The initial or main thread is simply a continuation of the process itself, and typically uses the original user-space stack. Stacks for subsequent threads are allocated on the system heap.

Many-to-one threads are fast: a context switch between threads is often under 10 times the cost of a procedure call, and creating a new thread is the cost of the malloc of the stack plus small change.

They can also be quite flexible: since they’re part of the application (usually as a

library), they can be customized by application policies quite extensively. The user-space threads package treats the underlying *process* basically as a dedicated, but virtual, processor.

The drawbacks of this design, however, come from the fact that the underlying process is not, actually, a processor, and a single process in this model can't actually take advantage of multiple real CPUs.

For example, what happens when a thread takes a page fault? The kernel is entered, and decides to block the process while it services the page fault, despite the fact that there might be other threads which could make progress.

Something similar happens when a thread issues a blocking system call (such as `connect()`). The kernel will stop *all* the threads in the process, since it doesn't really know about them.

For this reason, user-level threads packages using this model went to great trouble to "wrap" all blocking system calls with non-blocking equivalents which then called the user-level thread scheduler, but this adds quite a bit of complexity and *even so* doesn't handle the page fault case.

#### 4.12.2 One-to-one threads

The alternative to this approach is to simply have every thread in user space have a corresponding thread in the kernel, as in Figure 4.3. Now any user-level thread can be scheduled on any available processor, as many threads in the same program can run as there are free cores, and each thread can block independently without affecting the others in the same process. Indeed, a process now becomes an address space and a collection of threads.

Most mainstream operating systems now offer threads like this by default, including Linux, Solaris, Windows XP, and MacOS.

Note that this requires a kernel with multiple stacks (one per thread), and also requires most of the complexity of inter-thread communication and synchronization to be pushed into the kernel.

Since the kernel now creates and destroys threads, each one is allocated a separate area of the process' virtual address space for its stack, resulting in a memory layout shown in Figure 4.4.

This model offers excellent integration with the OS, but also has its drawbacks. For one thing, threads are slower: creating a thread is a Big Deal, almost as heavyweight as creating a new process. Context switch times between threads in the same application are almost as slow as switching between processes.

Also, note that the decision about which thread within an application to run is now taken by the kernel, rather than by the application library (as was the case with many-to-one threads). It's true the kernel now gets to see all the application threads, and which ones are blocked and which ones are runnable, but it has no idea which

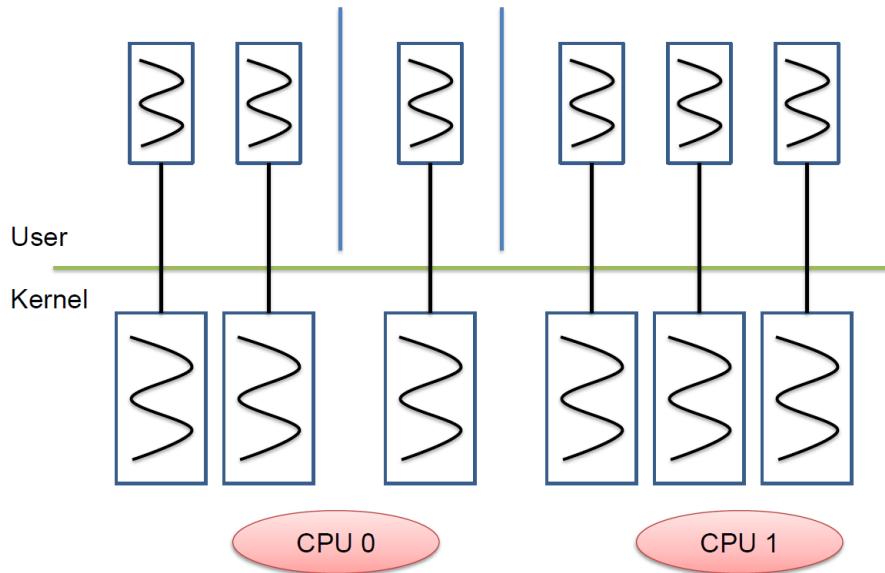


Figure 4.3: One-to-one (kernel) threads

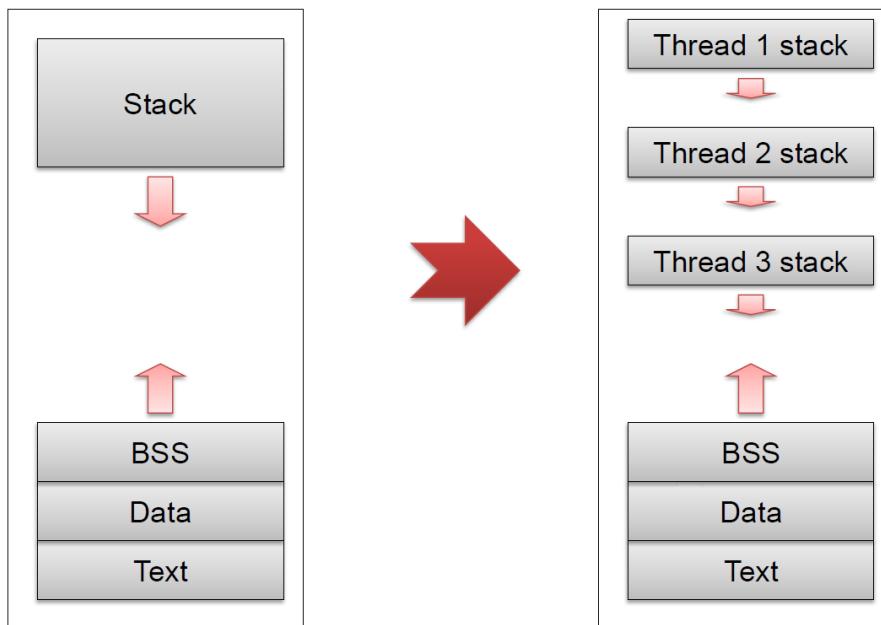


Figure 4.4: Memory layout for one-to-one threads

ones are more important to the application. Instead, it applies a one-size-fits-all scheduling policy and hopes for the best.

Finally, quite a lot of complexity has now moved into the kernel: all the machinery for creating, destroying, blocking, unblocking, and synchronizing threads must now run in privileged mode.

One might expect this to slow things down as well (due to more kernel entries/exits) and this is true, but not nearly as much as you might think. Modern implementations try to avoid entering the kernel as much as possible by trying all kinds of lightweight synchronization in user space before falling back to the kernel implementation (Linux “futexes” are one example), and applications work around the overhead of thread creation by creating a pool of threads in advance and not destroying them. Note, however, that the code complexity is still there in the kernel, although it's now invoked less.

Tellingly, the overhead and policy inflexibility of one-to-one threads has led many applications, language runtimes, libraries, and databases to ignore them: they create one thread per core in the system, and then implement their own threads on top of this fixed pool of “kernel” threads. Things are faster now, but the complexity remains, and all the problems with many-to-one threads return again.

### 4.12.3 Many-to-Many Threads

Ideally, we would like to combine the two models, as shown in Figure 4.5, and multiplex lots of user-level threads over several kernel-level threads.

In this model, we're using *kernel* threads to provide access to multiple physical processors at the same time and handle blocking I/O operations, and *user level* threads to abstract parallelism and concurrency, since they are cheap and fast. For performance or predictability, we can pin a user thread to a kernel thread and regain some control over scheduling policy.

Linux doesn't provide this functionality, but it is now the default in Windows (where user space threads are called “fibers”) and Solaris.

The key problems with this model, as with the others, is the lack of shared knowledge about the state of the system between the user space program (and library) and the kernel. It's hard for the runtime to tell:

1. How many user-level threads can potentially run at the same time (since this depends on how many physical cores the OS has decided to allocate to the process, and
2. Which user-level threads are actually running right now (i.e., which physical cores are currently running the process, as opposed to threads of another process)

This lack of shared knowledge severely limits the benefits of user-level scheduling, and can critically impact performance of parallel applications - which is why many

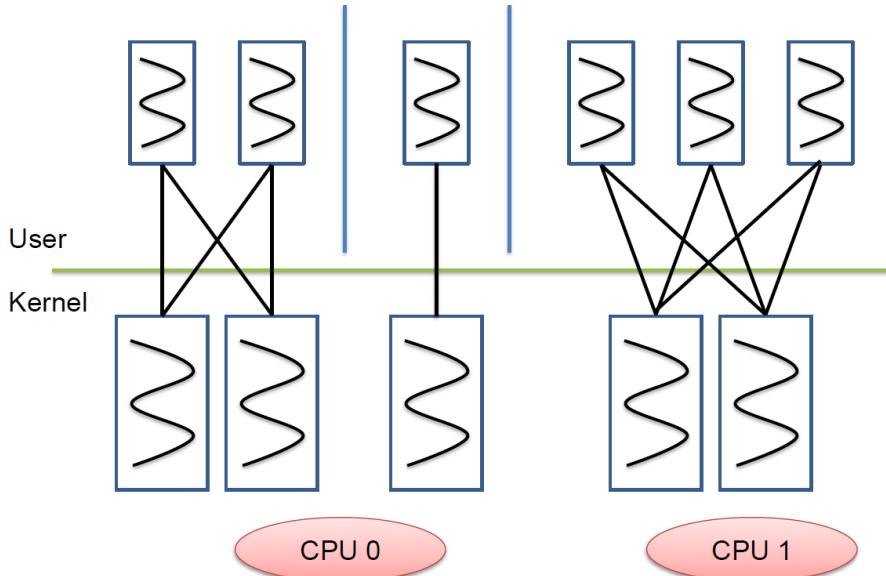


Figure 4.5: Many-to-many threads

High-Performance Computing applications go to great lengths to avoid anything else running at the same time.

There are solutions to this problem, but they involve rethinking thread scheduling. To understand them, it's worth making a distinction between *scheduling*, which is the process deciding which entity (thread, process, task, etc.) to run on a core at any given moment, and *dispatch*, which is the actually business of starting (or resuming) executing that chosen entity.

Two approaches to addressing the user-space thread problem were published at the same time (indeed, in the same presentation session at SOSP) in 1991, though one is rather better known due to its catchy name. Both worked by rethinking how *dispatch* works between the kernel and user processes. Barrelyfish adopts a solution influenced by both (though somewhat simpler).

#### 4.12.4 Scheduler Activations

Scheduler Activations [7] address the problem of the user-level scheduler not knowing what the kernel-level scheduler is doing by having the kernel tell it.

The basic mechanism is an incredibly powerful idea in systems called an *upcall* [33]: instead of the user program calling *down* into the kernel, the kernel calls *up* into the user program.

In the OS described in the paper, whenever the number of processors allocated to an application running on the system changes, the application is notified by an upcall: a designated function in the program's user-level scheduler is called.

The question that immediately arises is: called by what? Presumably not an application thread, since the application's threads are doing other useful stuff. Possibly a kernel thread, but the kernel threads are supposed to be running application threads, and we have just established that this upcall is not an application thread.

The context in which the user-level thread scheduler is entered is a new, specially manufactured thread-like kernel object called a scheduler activation. It is scheduled by the kernel like a thread, but only exists in order to provide an execution context for upcalling the user level scheduler. A new scheduler activation is created on demand in response to any kernel scheduler event: preemption, blocking, page faults, etc.), and used to notify the application that something has happened.

Naturally this requires the application to be running, and the target for this is multi-threaded parallel programs running on a multiprocessor, so in the common case some other part of the application was running on another core.

With the extra information provided by the upcall, the user-level scheduler could make smarter decisions about what to do with its threads and the paper showed some quite impressive performance improvements, even on a 7-processor Firefly VAX workstation.

#### 4.12.5 Threads in Psyche

Published at the same conference, the Psyche operating system [79] adopted a similar approach to Scheduler Activations, but with a slightly different perspective. Instead of viewing the activation as purely a way to notify the user program that its kernel core and thread allocation had changed, Psyche viewed the upcall as the user-space analogue of an interrupt in the kernel: the interrupt was an opportunity for the user library to change its scheduling of threads, block some, unblock others, and then run whichever thread it felt was appropriate.

In this way, the Psyche user-level scheduler was modeled a lot like an OS, with “hardware” provided by the kernel in the form of a shared area of memory divided into two regions: both readable by the user program, but only one half writable by it. This region was used to communicate information to and from the kernel, and worked like hardware registers. The upcall was even referred to as a “software interrupt”.

Upcalls interrupted the user program when a variety of events happened, including timer expiration, start of a blocking system call, completion of a blocking system call, and even “imminent preemption” – in other words, early warning that the program was about to be preempted. It's unclear if this latter facility was terribly useful. Curiously, page faults were *not* reflected to user space.

Figure 4.6 shows the data structures used in Psyche.

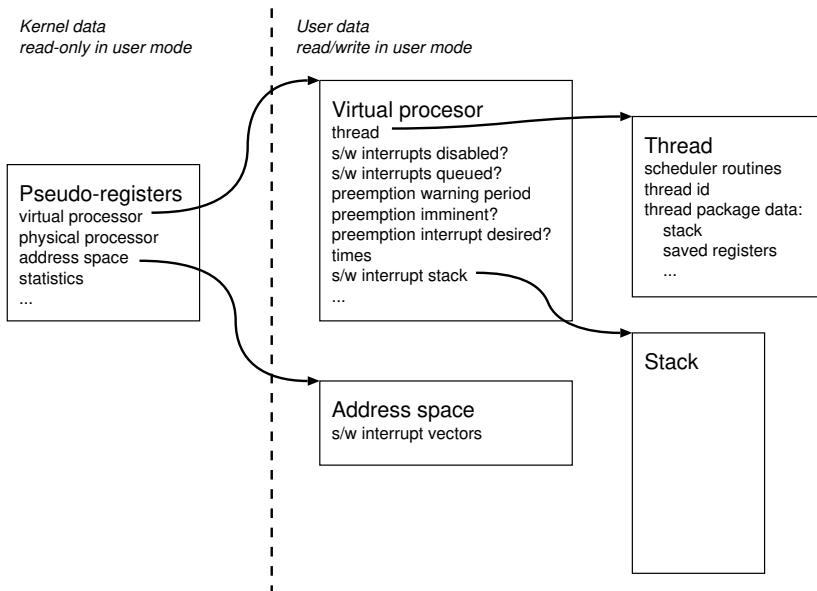


Figure 4.6: Data structures for Psyche threads

Psyche had several interesting features. Upcalls could be nested. They occurred on a dedicated user-space stack, and as long as this didn't overflow you could have several outstanding ones (though this complicated the user-level scheduler, which now had to be *re-entrant*). Upcalls could also be disabled or queued, much like hardware interrupts. The scheme was quite flexible, though it had the downside that a number of user-space data structures had to be pinned in memory.

## 4.13 A moment of Zen: threads and stacks

If you did **not** find the previous section confusing, you should probably go back and re-read it a couple of times.

As programmers, we tend to think of our programs at all times in terms of threads. What's cool, but also challenging, about operating systems is that it forces you to confront the fact that threads are an illusion created by more software (the OS and runtime). In order to implement threads (and processes), someone has to write code to create, destroy, and switch them, and when this code runs it is typically *not a thread*. It has registers, variables, instructions, and it may even have a stack, but it's not a thread.

Upcalls in Scheduler Activations, Psyche, and Barreelfish, are an extension of this idea into user space libraries, and these upcalls are also not threads.

People who aren't comfortable with this idea (and even many Linux kernel develop-



Commentary

ers fall into this category) are a very different kind of programmer from those who do.

We'll see more examples of this deconstruction of the thread abstraction later when we look at fast intra-core Inter-Process Communication (IPC), which is all about abusing the idea of threads in the pursuit of performance.

If you can grok all of this, you will have become a completely different, and much more insightful, type of programmer than the vast majority, and your perspective on all kinds of complex software systems will change radically – for the better.

As Chandu Thekkath once remarked:

*I would not trust my life to software written by someone who has never implemented an interrupt handler.*

If you *did* find the previous section confusing, then it's possible that the following concrete description of what happens in Barreelfish will help make it clearer.



Technical  
Details

## 4.14 Dispatch in Barreelfish

Barreelfish adopts an approach close to Psyche, but somewhat simpler (and descended from K42 [8] and Nemesis [72]).

Just as the Barreelfish CPU driver is non-preemptible and non-re-entrant, the Barreelfish user-level scheduler is also non-re-entrant, and upcalls are disabled while the user program is executing in the user-level scheduler.

Upcalls can be disabled by either the kernel or the user program by means of a flag (called the *resume bit*) in the shared writable area of the domain control block. Checkout the file `kernel/dispatch.c` and the calls to `resume()` and `execute()` respectively.

As in Psyche, when a Barreelfish process (or domain) is descheduled (either due to a kernel interrupt, page fault, blocking system call, or whatever), you can think of it as having experienced the user-mode equivalent of an interrupt. Just as with a “real” interrupt, the context (registers, program counter, etc.) are saved in an area of memory, and the appropriate handler is called.

By analogy with kernel-level interrupts, it is the responsibility of the user space program to reload the context from this saved information when it decides to resume the thread. However, it could equally well decide to resume the saved context of a *different* thread to implement a context switch.

Here's how things work in practice. The Barreelfish dispatcher control block is divided into two sections, one of which is only readable by the user program, and the other of which is writable as well. In the read/writable section are two areas in which the processor context (registers, etc.) can be saved. We call these the *activation slot* and the *resume slot*. There is also a word holding the address of the first instruction of the upcall handler, the *activation address* (a bit like an interrupt handler).

Suppose the application is running, and the kernel is suddenly entered (for example, the hardware timer interrupt goes off). The kernel saves the execution state and enters the kernel scheduler as follows:

```

1 if resume bit == 0:
2   activation slot ← processor state;
3 else:
4   resume slot ← processor state;
5 schedule()

```

Note that *where* the process register state is dumped depends on whether upcalls are enabled or disabled when the interrupt occurs. If they were disabled, the state is saved in the *resume slot*. If they were enabled, the state is saved in the *activation slot*.

So far so good. Later, when the CPU driver decides to dispatch the process again, this happens:

```

1 if resume bit == 0:
2   resume bit ← 1
3   jump to (upcall) the activation address
4 else:
5   processor state ← resume slot

```

If the resume bit is 1, the processor state is simply resumed from the resume slot. This mimics the behavior you're familiar with from Unix or Windows: the process just continues where it left off.

However, if the resume bit is 0, then something different happens: the CPU driver instead *upcalls* the user level scheduler, having set the resume bit to 1.

This upcall isn't a thread (like Anderson et al, we call it an activation), and it even has its own stack which is also part of the read/writable area of the Dispatcher Control Block (DCB).

This means that the user level scheduler now runs, and won't be disturbed by further upcalls (since the resume bit is 1) until it decides which user-level thread to run. It then clears the resume bit and dispatches the thread. If there were any pre-emptions before this happens, the user-level scheduler won't notice them since the CPU driver will simply resume it where it left off.

This approach turns out to have pretty much all the advantages of Psyche or Scheduler Activations in providing the user-level scheduler with all the information and timing it needs to efficiently manage user-level threads, but ends up being somewhat simpler as long as the kernel is similarly simple (as it is in Barrelyfish).

It's also the reason why you'll sometimes hear us proudly claim that Barrelyfish has no kernel support for threads, since all it really deals with are dispatchers.



## 4.15 Challenging hack: resuming a thread

It turns out that clearing the resume bit and resuming a thread from a context previously saved after an interrupt is a bit harder than it sounds. There are two tricky things to get right.

### Atomically clearing the bit

The first trick is to ensure that the resume bit is cleared and the thread is resumed atomically. Obviously you can't clear the bit *after* the thread is resumed (since you're now running the thread's code instead), but you also can't clear it before the thread is resumed unless you can guarantee that the code afterwards isn't preempted until the thread is fully resumed. This race condition is inherent in implementing a user-level thread scheduler over upcalls.

Both Psyche [79] and the original scheduler activations implementation [7] (as far as we know) allowed this to happen by pushing the context on a stack. This meant that the user-level thread scheduler had to be *re-entrant*, and also required there to be not too many preemptions, otherwise the scheduler would run out of stack.

The first implementation of Nemesis [72] targeted DEC Alpha processors [5], and used a wonderful feature of the Alpha architecture called PALcode. PAL stood for *Privileged Architecture Library* and was somewhere between true microcode and kernel mode: in “PAL mode”, all interrupts and the MMU were off, many interlocks were disabled (i.e., you had to handle pipeline delay slots yourself) and a large number of micro-architecture-specific registers were available. Otherwise, it was regular Alpha assembly language. Entering and exiting PALmode happened on a trap, reset, TLB miss, or the execution of a special *PALcall* instruction, and only cost a single pipeline drain.

The idea was that a different PAL image was written for each OS, and each model of processor, and adapted the particular processor to the current OS at hand. DEC supplied PALmode images for OpenVMS [51], Windows NT [35] (which ran on Alphas at the time), and OSF/1 (DEC’s version of Unix for Alpha, based on Mach 2.5).

PALcode was a truly great hardware feature from a kernel designer’s perspective, and is fondly remembered. The PALcode image written for Nemesis provided a very fast PALcall to resume a thread and clear the resume bit atomically.

On other processors, this atomic clear-and-resume is a bit tricky. Ports of Nemesis used a small (but potentially still expensive) system call to do this. Very early versions of Barrelfish for 64-bit Intel Architecture machines also adopted this approach, until Justin Cappos suggested an alternative: when a process is preempted, the CPU driver not only looks at the current value of the resume bit, but also if the program counter is within the bit of code which resumes the thread. If the latter is true, it treats the resume bit as set *regardless* of its actual value. In this way, the thread resume code works by entering the special section, clearing the bit, and re-

suming the thread (which causes the processor to jump out of the special section). Barrelyfish now uses this technique for most target architectures.

### Resuming the thread

The second challenge sounds easier: simply resuming the thread. More precisely, the problem is this: given a saved processor context written by the CPU driver to save the process state following an interrupt, resume the process *from user space code*, i.e., without transitioning between kernel and user mode, or using any privileged instructions.

On the Alpha processor, this was all handled by PAL code.

On CISC 64-bit Intel Architecture PCs, it turns out that the `iret` instruction to return from an interrupt can be called from user space [59]. The sequence is a bit fiddly, since it involves pushing the last few context values (condition codes, stack pointer, program counter, etc.) on the stack before immediately popping them off with `iret`, but it works (and with caches on a modern processor) it's fast:

```

1 mov    %[fs], %%ax
2 mov    %%ax, %%fs
3 mov    %[gs], %%ax
4 mov    %%ax, %%gs
5 movq   0*8(%[regs]), %%rax
6 movq   2*8(%[regs]), %%rcx
7 movq   3*8(%[regs]), %%rdx
8 movq   4*8(%[regs]), %%rsi
9 movq   5*8(%[regs]), %%rdi
10 movq  6*8(%[regs]), %%rbp
11 movq  8*8(%[regs]), %%r8
12 movq  9*8(%[regs]), %%r9
13 movq  10*8(%[regs]), %%r10
14 movq  11*8(%[regs]), %%r11
15 movq  12*8(%[regs]), %%r12
16 movq  13*8(%[regs]), %%r13
17 movq  14*8(%[regs]), %%r14
18 movq  15*8(%[regs]), %%r15
19 pushq %[ss]
20 pushq 7*8(%[regs])
21 pushq 17*8(%[regs])
22 pushq %[cs]
23 pushq 16*8(%[regs])
24 movq  1*8(%[regs]), %%rbx
25 iretq

```

32-bit ARMv7 [11] doesn't allow this, but it does have two important (and unusual) features. Firstly, the program counter is a regular processor register (`r15`). Secondly, there is an instruction – `ldmia` – which atomically loads multiple registers from memory in one go. Hence the following code:

```

1 clrex
2 mov r2, #0
3 str r2, [r0,#disabled_flag_offset]
4 ldr r0, [r1], #4
5 msr cpsr, r0
6 ldmia r1, {r0-r15}
7 mov r0, r0

```

The final `mov r0,r0` instruction does nothing, but deals with the one-cycle delay before the value of `r15` triggers the jump.

So far so good. What about other architectures? ARMv8 [12] cleaned up the ARM architecture in many ways. However, at first sight, there is no obvious way to resume a thread from user space: a jump can only occur by loading the target address into a register, but if you're resuming a thread, you need that register to have been restored from the saved context when you take the jump, otherwise you've trashed one of the thread's registers.

You could *reserve a register* purely to use in resuming threads, and hope that register renaming in a modern core design means that little performance is lost by having fewer register names available. However, this breaks the standard AARCH64 ABI and calling conventions, and also feels very ugly.

An alternative would be to implement thread resume as a system call (which at least is *fairly* fast on most ARM processors), but kind of defeats the object of user-level context switching. If you can think of a better way to do this, please let us (and ARM Ltd.) know.

This is an example of processor architects thinking inside a very small box when it comes to architectural features: since neither Linux nor Windows need to resume a thread in user space, why would anyone else? User-level interrupts are actually planned for the ARM architecture (for virtual machine support), and they will allow a solution much like the one for (ironically) 64-bit Intel Architecture above.

In Barreelfish, we go for the “reserved register” option. Here’s the code which reserves the `x18` purely for the job of resuming a thread:

```

1 /* Restore PSTATE , load resume
2 * address into x18 */
3 ldp x18 , x2 , [x1 , #( PC_REG * 8)]
4 /* Set only NZCV . */
5 and x2 , x2 , #0 xf0000000
6 msr nzcv , x2
7 /* Restore the stack pointer and x30. */
8 ldp x30 , x2 , [x1 , #(30 * 8)]
9 mov sp , x2
10 /* Restore everything else . */
11 ldp x28 , x29 , [x1 , #(28 * 8)]
12 ldp x26 , x27 , [x1 , #(26 * 8)]

```

```

13 ldp x24 , x25 , [x1 , #(24 * 8)]
14 ldp x22 , x23 , [x1 , #(22 * 8)]
15 ldp x20 , x21 , [x1 , #(20 * 8)]
16 /* n.b. don 't reload x18 */
17 ldr x19 , [x1 , #(19 * 8)]
18 ldp x16 , x17 , [x1 , #(16 * 8)]
19 ldp x14 , x15 , [x1 , #(14 * 8)]
20 ldp x12 , x13 , [x1 , #(12 * 8)]
21 ldp x10 , x11 , [x1 , #(10 * 8)]
22 ldp x8 , x9 , [x1 , #( 8 * 8)]
23 ldp x6 , x7 , [x1 , #( 6 * 8)]
24 ldp x4 , x5 , [x1 , #( 4 * 8)]
25 ldp x2 , x3 , [x1 , #( 2 * 8)]
26 /* n.b. this clobbers x0&x1 */
27 ldp x0 , x1 , [x1 , #( 0 * 8)]
28 /* Return to the thread . */
29 br x18

```

RISC-V [106] is intended to be a new, clean, modern, flexible, and open RISC architecture, and for the most part it succeeds. However, it seems to suffer from the problem as ARMv8, for the same reason. However, Andrew Waterman at U.C. Berkeley devised the following hack:

1. Find a register whose value never changes while a thread runs, and which holds the address of a user-readable data structure. Such a register is gp, the “global pointer”, which points to a process’ global object table (GOT).
2. Arrange to place some machine code immediately *below* the GOT in memory
3. This code first restores all the thread context *apart* from gp.
4. Then, it loads gp with the resume address, i.e. the target thread’s program counter value to jump to.
5. Finally, the last instruction (just before the start of the GOT) is a jump-and-link to the address held in gp, with the destination register for the return address *also* gp. This restores gp to the correct value (the address of the GOT) and executes the jump at the same time.

The key bit of the code looks like this:

```

1 lw gp, offset(tp)
2 jalr gp, gp, 0
3 gp: <global data>

```

This code is either very beautiful or horribly ugly, depending on your perspective.

Tragically, the “Waterman hack” for RISC-V doesn’t work on ARMv8, since the ARMv8 jump-and-link instruction is hardwired to store the return address only into the link register LR and no other.

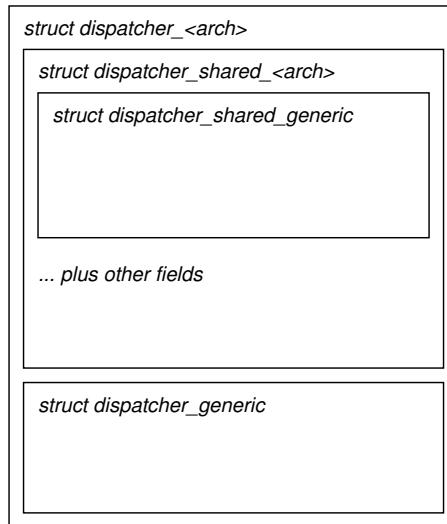


Figure 4.7: Dispatcher structure. `<arch>` is arm, aarch64, etc.

The moral of this digression is this: processor architecture is often unconsciously influenced by OS orthodoxy, and not always in a good way.



## 4.16 Set up the dispatcher

In the Barrelyfish CPU driver, all information about a process (on a core, effectively a dispatcher) is represented by a capability type called a dispatcher control block or DCB.

It is created by the spawning process and then given to the CPU driver. The CPU driver will use this memory area to store information about the process. For example, it will save information like register state on a context switch, or the error handlers to be called on page faults.

The size of the dispatcher frame is given by `DISPATCHER_FRAME_SIZE`. You should allocate a frame of this size and map it into both `init`'s and the child's address space. The capability for this frame should also be stored in the child's CSpace in the appropriate slot.

The dispatcher structure itself is composed of several different types of fields, as shown in Figure 4.7, depending on whether the field is the same for all architectures (`dispatcher_generic`), or it only exists for a specific processor architecture (`dispatcher_shared_<arch>`), or (the third case) the field exists on all architectures but with a different format for each architecture (`dispatcher_shared_generic`).

The dispatcher contains two register save areas enabled and disabled, which

are used to implement the “scheduler activations” upcall discussed above, and help with user level thread-scheduling. The process should start in disabled mode. To get access to these structs you can use the following code, assuming you have a pointer to the mapped dispatcher frame stored in `handle`:

```

1 struct dispatcher_shared_generic *disp =
2     get_dispatcher_shared_generic(handle);
3
4 struct dispatcher_generic *disp_gen = get_dispatcher_generic(handle);
5
6 arch_registers_state_t *enabled_area =
7     dispatcher_get_enabled_save_area(handle);
8
9 arch_registers_state_t *disabled_area =
10    dispatcher_get_disabled_save_area(handle);

```

We need to fill in some initial information in the frame so when the CPU driver will try to switch to the program for the first time, it will not immediately crash. The information you must fill in is as follows:

```

1 // core id of the process
2 disp_gen->core_id = ... ;
3 // Virtual address of the dispatcher frame in child's VSpace
4 disp->udisp = ... ;
5 // Start in disabled mode
6 disp->disabled = 1;
7 // A name (for debugging)
8 strncpy(disp->name, ..., DISP_NAME_LEN);
9 // Set program counter (where it should start to execute)
10 disabled_area->named.pc = ...;
11
12 // Initialize offset registers
13 // got_addr is the address of the .got in the child's VSpace
14 <arch> _set_registers(got_addr, handle, enabled_area, disabled_area)
15
16 // we won't use error handling frames
17 disp_gen->eh_frame = 0;
18 disp_gen->eh_frame_size = 0;
19 disp_gen->eh_frame_hdr = 0;
20 disp_gen->eh_frame_hdr_size = 0;

```

## 4.17 Set up arguments

Next, we deal with the command-line arguments to the new process. If the process is started by an actual shell (and this may well be the case later in the project), the shell needs a way to pass such arguments to the new process.



Project  
Instructions

At the beginning of the arguments page (i.e., the frame in `TASKCN_SLOT_ARGSPAGE`), the initialization code assumes that there is a `struct spawn_domain_params` lying there. The child's startup code expects everything that does not explicitly have to be filled in by `init` to be zeroed. The remainder of the memory page (i.e., after `struct spawn_domain_params`) is used to store your command line arguments. Make sure that no pointer stored inside your `spawn_domain_params` points into `init`'s address space, but instead all should refer to the child's address space. Also, make sure that after the valid `argv` and `envp` pointers, there is a NULL pointer to signify the end of the list. The child's bootstrap thread expects a pointer to the `struct spawn_domain_params` as its argument (in its own virtual address space of course). So you need to make sure that the register for the first argument in the enabled save area contains this pointer, such that when the register state is restored and the function called, the argument is passed properly. You can set the register using the `registers_set_param` function.

To modify arguments for a process after compilation time, pass on the arguments from the multiboot specification. Then, the arguments for your new process can be specified in the `menu.lst` file. You can access these arguments using the `multiboot_module_opts` function. Since this function will only return a string, you will have to split it into multiple arguments. Implement at least space separated command line arguments. Optionally, you can implement a more sophisticated parsing method (such as escaping with backslash or quotes). To ensure the argument passing works, modify the `hello` process so that it prints after the hard-coded message its first command line argument.



*Project  
Instructions*

## 4.18 Start the process at last

Finally, you can now make your dispatcher runnable by invoking the dispatcher capability using the function `invoke_dispatcher`. As `domdispatcher` you can pass `cap_dispatcher` that is statically defined.

If you've done everything right, the child process should be able to print a message. If so, congratulations, you've created a process and successfully started it!



*Commentary*

## 4.19 Note on Hakefiles

Inspecting the Hakefile for the `init` process, you may have noticed the additional link flags present for this target:

```

1 [
2 build application { target = "init",
3 cFiles = [ "main.c" ],
4 -- link flags
5 addLinkFlags = [ "-e__start_init" ]
6 ]

```

This overrides the default start symbol `_start` to `_start_init`. Have a look at the architecture specific assembly file `crt0.S` in `lib/crt` and try to understand what exactly happens there. In short, you can use the different entry points to distinguish whether the process that has just started is `init` or an other process. Thus, for other domains than `init` you can omit the `-e _start_init` link flags.

## 4.20 Process Management

Congratulations, you now managed to spawn a few new processes. Operating systems usually have functionality to list the processes running and provide a mechanism to (forcefully) terminate them. On Linux, you can execute `ps -aux` to display a list of processes, and `kill -9 <PID>` to send the kill signal to the process with the supplied PID.

For now, we are not going to implement full process management functionality. However, you should implement some of the mechanisms and bookkeeping necessary to implement this functionality later.

First, this entails to keep track of the processes you've spawned. This will be some data structure that contains information about the processes such as its name, PID, and specific capabilities (e.g., the dispatcher capability). Later, you can add other information to this data structure such as the endpoint capability or the RPC channel. The data structure should offer an API to add/remove/lookup/enumerate processes. You don't need to worry about keeping the list up-to-date automatically in response to processes terminating as of yet.

Secondly, implement a mechanism to (forcefully) stop the execution of the process, i.e., remove the dispatcher from the run queue. One way to demonstrate this is to spawn a process that prints something every second, and then kill it. There are multiple ways to achieve this. Hint: each running domain must have a dispatcher in the run queue. Recall, Barreelfish's capability system relies on a user-space domain called the *monitor* that is involved with distributed and more complex capability operations such as revoke or cleaning up certain capabilities such as CNodes and the dispatcher. This is because deleting the *last* copy of a CNode or dispatcher capability may trigger a recursive deletion of other capabilities. The monitor breaks this down into delete steps (see the individual milestone [chapter 13](#)). Thus, revoking or deleting the last copy of the dispatcher will result in a failure. What can you do?

To stop the execution of a domain, we are going to *add* a new operation to the dispatcher cap. This involves three things:

1. Define a new command `DispatcherCmd_Stop` for the dispatcher cap.
2. Implement an invocation function `invoke_dispatcher_stop` that takes a dispatcher capability as its single argument.
3. Implement the corresponding handler functions in `arch/armv8/syscalls.c` and in `syscalls.c`.



Project  
Instructions

Hint: you may have a look at `invoke_dispatcher_properties`.

## 4.21 How to do well

The following hints might be useful:

- Your `init` process has only a 64kB stack. Make sure that you do not put any big data structures there – use the heap.
- It's okay to use `malloc` and `free` for your implementation, but be aware that the libaos provides `init` only with a static heap size of 16 MB (see `lib-aos/morecore.c`). If this isn't enough, you can use the slab allocator you already encountered in the previous milestone.
- After your child process terminates, the kernel might print "revoking dispatcher failed", don't worry about that (yet).

You should be able to explain each of the steps for creating a new process, and how each of them works.

Also, you should understand how you setup the child's CSpace (and why!), and its VSpace (creating the page table, mapping the binary file's sections, etc.).

Make sure that you can not only start a child process, but that you can do this more than once.



*Commentary*

## 4.22 Milestone 2 Summary

Milestone tasks:

- Fully implement `paging_alloc` and `paging_map_frame_attr`.
- Show that by using `spawn_load_by_name` you can start multiple child processes by printing a message from the new processes.
- Show that you can stop the execution of the child process.



*Project  
Instructions*

Extra challenges:

- Your child process is unaware of its `struct paging_state`. A simple way to avoid running into trouble with such a limitation is to prevent mappings within certain regions of the virtual address space. Alternatively, to allow arbitrary mappings in the child, the complete `paging_state` must be passed to the child. Implement a method for passing the `struct paging_state` to the child.
- Implement the unmapping functions such as `paging_region_unmap`.

Assessment:

- Show your implementation of `paging_map_frame_attr` is correct by mapping a large frame.
- Show that you mapped in the ELF binary correctly by printing its magic number.

- Be able to explain how you set up the CSpace and VSpace and where the child's capabilities are stored.
- Show that you can start a process by letting the new process print a message.
- Start the process multiple times.
- Ability to stop the process again.

# Chapter 5

## Message passing

By now, your operating system should be able to manage physical memory and spawn new user-space processes. To get any further, you need to be able to communicate between processes, and this is the theme of this week's assignment.

The particular flavor of IPC that you will be working towards for the rest of this milestones is Remote Procedure Call (RPC). You can think of RPC as a procedure call that happens to call into another process. RPCs provide a uniform interface on which you can implement server-client communication with different system services that provide functionality such as physical memory management.

In this assignment, you will implement the core RPC mechanisms for both client and server side, and you will implement and use the first services: the memory server, and serial output over RPCs to `init`. The functionality required is:

- Simple RPC communication between two initial domains
- Passing capabilities between the domains
- Invoking a memory server in the `init` process
- Using `init` as serial driver.

Please note that you will have to implement more RPCs and servers in future milestones, so do take some time to think about a suitable design which allows you to easily implement new RPCs and new RPC servers in the future.

### 5.1 The Basics

You're probably familiar with a few different methods in Unix for two processes to communicate with each other. Unix systems have a lot: pipes, signals, Unix-domain sockets, POSIX semaphores, FIFOs (also known as named pipes), shared



Commentary

memory segments, System V semaphore sets, POSIX message queues, System V message queues, and probably several more. Windows actually has rather more.

In contrast to Unix, in this chapter we'll also look at how later versions of TAOS, L4, and Barrelfish implemented IPC. Many other variants exist, so it's worth mapping out the design space a bit for IPC designs. There are plenty of choices to be made:

- Is IPC based on one-way *messages*, two-way *remote procedure calls*, or a combination of the two?
- Is the communication *blocking* or *non-blocking*? Sometimes an operation cannot complete immediately. A send might not be able to finish because there's no space to buffer the message. A receive might find there is no message or call waiting to be received yet. Does the thread performing such a send or receive operation block, or return immediately with an error?
- Is more than one thread involved? Does a single thread move between processes or address spaces as part of the communication, or do threads at the sender and receiver exchange information?
- How is the CPU scheduler involved? To what extent does sending a message cause a thread to become blocked or unblocked?
- What is the difference (in implementation) between the case where the communicating processes are on the same core, and the case where they happen to be running on different cores?
- Is *notification*, in other words telling the destination process that communication has happened, decoupled from *data transfer*, which is actually conveying a message payload between communicating entities?
- What is the programming interface? Is it based on sending strings of bytes (as in networked communication), or is it trying to move data from CPU registers in one process to other registers in another?

In Unix and Windows, IPC is message-based and usually heavyweight – think of copying data through pipes or Unix domain sockets. It is usually *polled*: receiving a message is typically done with a blocking call (such as `read()` or `recv()`). These calls tend to combine notification, scheduling, and data transfer in a single operation. Non-blocking I/O with `select()` or `poll()` can be used to separate data transfer but still combines notification and scheduling.

Performing notification independently of data transfer and scheduling requires some interrupt-like mechanism in user space – in other words, an upcall. Unix does have a very limited upcall facility: signals. Signals execute on whatever user-space stack happens to be active when the signal is delivered (contrast with the upcall description in the previous chapter), and can only safely execute a limited set of system calls. Calling out of a signal handler with `longjmp()` is an adventure. In general, Unix lacks a good upcall / event delivery mechanism.

Other OS designs can be very different, as we shall see. We first consider whether

IPC is one-way or two-way - RPC can be implemented *above*, say, Unix pipes, using a suitable stub compiler, but other systems have provided it as a basic primitive.

## 5.2 Local Remote Procedure Call

In a classical RPC system [29], invoking a function on a remote machine follows a familiar pattern:

- When user client code invokes the local stub of a remote function, this stub first:
  1. Marshals arguments to the remote procedure into a buffer
  2. Sends this buffer over the network
- The server then:
  1. Receives the buffer
  2. Decodes which procedure is to be invoked
  3. Unmarshals the arguments to this procedure from the buffer
  4. Calls the procedure
  5. Marshals the return values from the procedure into a buffer
  6. Sends this buffer back to the sender
- Finally, the client:
  1. Receives the buffer
  2. Unmarshals the return values from the buffer
  3. Returns these to the client function which invoked the RPC.

Rather little has changed since Birrell and Nelson's day – this is still pretty much how remote invocations are made in web services, protocol buffers, you name it. What is called is usually referred to as a “method” these days, but note that even in the original RPC formulation, the concept of an object is implicitly present: the client is calling a named procedure *on a particular server*. It works.

Some details are missing from the description above. One is access control: the server may want to check that the client is allowed to call the given procedure. Another is the subtleties of the programming interface, such as whether a client process can have more than one outstanding invocation in process at the same time, or if requests can be restarted.

Another question is what happens when things go wrong. A big difference in RPC programming, often forgotten by enthusiastic programmers discovering it for the first time, is that things can *and do* go wrong in ways that can't happen in regular



Background

procedure calls. Jim Waldo et al. wrote one of the clearest articulations of this difference [105].

These concerns aside, RPC has a lot going for it as a clean abstraction, and many operating systems (including Windows) have adopted it as a basic communication primitive even for *local*, inter-process interactions on a single machine – hence the somewhat oxymoronic term “local remote procedure call”.

### 5.2.1 Overhead of local RPC

One of the first systems to use RPC locally, indeed to structure most of the OS around it, was the TAOS environment for the experimental Firefly workstation at the DEC Systems Research Center in Palo Alto in the 1980s (this is perhaps not surprising since Andrew Birrell was part of the project). One of the first things the group noticed was that RPC seemed to be slowing the system down.

Mike Schroeder and Mike Burrows published a detailed analysis of the performance of local RPC on the Firefly hardware [94], and it’s a classic paper. They pointed out quite a large number of factors contributing to the high latency of RPC calls, and also the low throughput possible in the system:

- Stub code copies lots of data from one area of memory into another – one of the two being the RPC buffer.
- Inside the kernel, data was copied too since each process had its own buffers for communicating with the kernel. In many cases, between different processors in the system data could be copied 4 times in the course of a single RPC.
- Much time was spent on access validation: after a client thread had tried to perform a call, the kernel spent significant time trying to figure out if this was allowed.
- Messages were queued waiting for transmission and reception in the kernel, as with network packets. The enqueue and dequeue operations cost valuable machine cycles.
- Significant time was spent in the OS scheduler getting two threads to rendezvous. While the appealing feature of RPC to programmers is that it *appears* as if a single thread of control has crossed in to the server’s domain, and then returned with the results, in practice the TAOS scheduler had to transfer message payload between two threads and block/unblock them both.
- Each RPC took 4 half-context-switches – entering or exiting the kernel. It also involved a processor swapping from one thread to another at least twice (once for call, and once for return). When the RPC was handled by a different processor, this could actually be higher, since you could not simply context switch from the client to the server and back.

One might ask why this wasn't the case with network RPC, which by then had been in fairly widespread use (initially at Xerox PARC, but thereafter in a wide range of early distributed systems).

The answer is that the overhead had always been there, but nobody had ever cared about it since the time taken to perform a truly *remote* RPC was dominated by the propagation delay in the network. Put simply, if the message takes 10ms to get from one machine to another, a software overhead of 100 $\mu$ s is neither here nor there.

## 5.3 Lightweight Remote Procedure Call



Background

The solution to this problem, Lightweight Remote Procedure Call (LRPC) [26], kicked off a fashion for high-performance IPC which even continues to this day. The goals were the following:

**Simple control transfer:** the client's thread in LRPC executes in the server's domain. In other words, there is no longer a need for a thread rendezvous.

**Simple data transfer:** LRPC uses an argument stack shared between client and server, plus processor registers if it can.

**Simple stubs:** in other words, highly optimized marshaling

**Design for concurrency:** LRPC avoids shared data structures wherever possible, to reduce contention and locking.

There were two main insights that were exploited in achieving this design. The first is that most messages are short – and Bershad et al. showed this by instrumenting a running system, collecting lots of data, and plotting the graph [26]. The second is that in most cases where you cared about performance, the client would call the server repeatedly, and so anything you could do once, in advance, rather than every time the procedure was called, would increase performance.

For this latter reason, LRPC has an explicit concept of *binding*. The word “binding” can have two related meanings. First, it is the process of setting up enough state in advance so that each subsequent call works – think of it as the distributed systems equivalent of connection setup in networking. Second, it also means the state that results from the binding process, which in some systems is a first-class object itself.

LRPC uses a model which has been replicated many times since, with some variations and often different terminology. For each procedure in a server which can be called from a client process, a Procedure Descriptor (PD) is registered with the kernel – this identifies something that a client can establish a binding to.

When a client wants to bind to an interface, a dedicated argument stack or *A-stack* is allocated for each procedure in the interface and mapped read/write into *both* the client and server address spaces. The kernel also allocates linkage records for return from each A-stack. The list of A-stacks is returned to the client as a capability to a *binding object*.

All this happens before any procedure call occurs, and only needs to happen once. Thereafter, each time a client wishes to invoke a procedure call on a binding object, the following happens (all on client thread):

1. The kernel verifies that the binding is valid and locates the correct PD.
2. It then verifies that the A-stack is valid, and finds the corresponding linkage
3. It ensures no other thread is using that A-stack/linkage pair
4. The caller's return address and stack pointer are put in the linkage, chaining the A-stack back to the thread's regular stack.
5. The linkage is then pushed onto a stack on in the thread control block (to allow for nested calls).
6. Next, the kernel finds an *execution stack* or E-stack in the server's address space. This is really just a stack, not a thread, and so requires no thread switch.
7. The E-stack is linked to the A-stack to make arguments available to the server.
8. The thread's stack pointer is adjusted to point to the E-stack
9. The virtual address space is now switched to the server domain
10. The server is then *upcalled* at a stub address specified in the PD to start the call.

The return path essentially consists of reversing the procedure above by unwinding the E-stack, then the A-stack, back to the client's original stack.

Very little locking is required here (and what contention there might be is mostly restricted to a single binding or server), and there's almost no copying of arguments due to the use of the A-stack. Quite a lot of work has been moved out into the process of creating a PD, A-stack, and E-stack in advance, none of which contributes to the overhead of a call.

There are other optimizations described in the paper (particular with regard to how the stubs are implemented), but the key idea is setting things up in advance, and optimizing not for large network buffers (as in classical RPC) but for the small number of arguments typical for procedure calls and which can be passed in registers or on the stack.

The dominating factor in the remaining cost is the address space switch, which is hard to improve in software – it's mostly hardware-limited, particularly as the VAX had no Translation Lookaside Buffer (TLB) tags.

As an aside, as described the approach works on a single processor, and carries the client thread itself across the protection domains. If you have more than one processor (as the Firefly did), the kernel checked for a processor idling in the server domain. If this was the case, it simply swapped the calling and idling threads

between processors (the Firefly had very small caches, so migrating a thread between cores was cheap).

## 5.4 IPC in L4



*Background*

Many subsequent systems adopted and extended these kinds of ideas, but the idea of optimizing the hell out of *local* remote procedure call between processes was probably pushed furthest in the L4 microkernel [76]. L4 has a reputation (somewhat deserved) for a maniacal obsession with Null-RPC latency: the time take to execute a remote procedure call between two processes which does nothing, and takes place while nothing else is happening (or runnable) on the machine.

There is a reason for this: in a microkernel, IPC performance is critical to overall system performance. Everything in early versions of L4 was designed around this metric, and in particular minimizing the number of instructions executed and number of cache misses required. L4 implementations for a given architecture remain the gold standard for same-core IPC.

IPC in L4 is synchronous: threads block waiting both for a reply to arrive and for the ability to deliver a message to the other side. IPC is therefore always a *rendezvous* between two threads, the sender and receiver. The sender blocks until the receiver wants to receive, and a receive blocks until a sender sends something.

Moreover, messages in early L4 are sent to threads, not any particular end-points. In other systems, this is made a bit more flexible: an *end-point* is a distinct object where senders and receive agree to rendezvous.

L4 avoids any LRPC-style kernel-allocated A-stack: since the receiver must have a waiting thread, a message operation simply exchanges register contents between the sending and receiving threads and carries on.

This design couples notification, transfer, scheduling, to the extent that it *is* the scheduler – the regular priority-based L4 scheduler is completely bypassed during an IPC operation, which is a somewhat controversial choice. However, it's very fast in the sense that it probably delivers latency as close to a theoretical lower bound as you can get.

### 5.4.1 Combining IPC operations

A further optimization in L4 is to combine pairs of IPC operations into a single call – in many cases, two IPC operations are executed by a client or server back-to-back. L4 provides only a single system call for all operations, and allows both the “sender” and “receiver” for the call to be specified, left blank, or (in the case of sender) given as a wildcard. In practice, there are 5 use-cases that are interesting, and they are shown in Figure 5.1. All are synchronous and unbuffered, to minimize latency.

**Send** is straightforward, and simply sends a message to a specified destination. It's equivalent to a `sendto` call in BSD sockets.

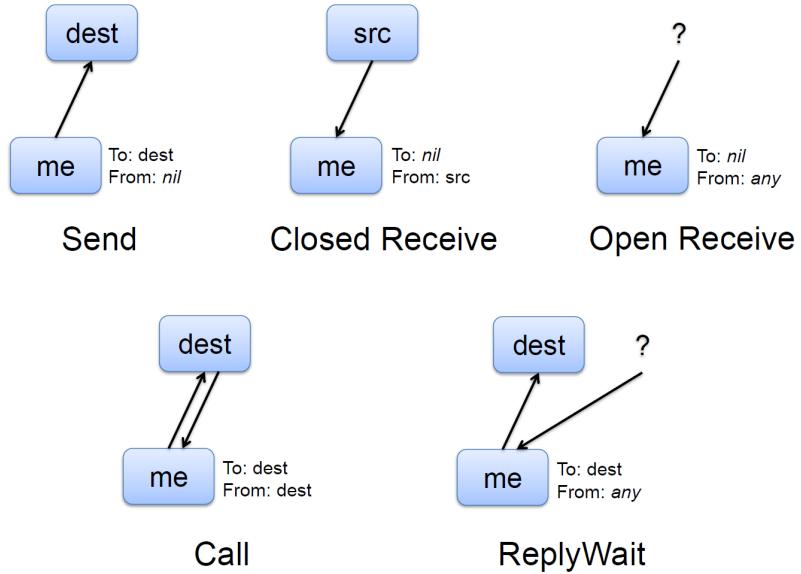


Figure 5.1: Combined IPC operations in L4

**Closed Receive** receives a message from a specified sender, much like `recvfrom`.

**Open Receive** will accept an incoming message from any sender.

**Call** is a combination of Send and Closed Receive; it sends a message to a destination and then waits for a reply to come back. This corresponds to the client of an RPC request.

**ReplyWait** is unusual: it sends a message to a specified destination, and then waits for a message from any source (combining Send and Open Receive). This is actually the server equivalent of Call: a typical server loop will start with an Open Receive to obtain the first client request, but thereafter will reply with the result of a request using ReplyWait in order to obtain the next request in the same operation.

Note that a more traditional network-oriented API like Berkeley Sockets has no equivalent of Call or ReplyWait. For a (slow) network, this is understandable since the overhead introduced by having to make two system calls is assumed to be small compared with the latency introduced by the network stack.

However, even in the network case, it's not clear this is the case any more. End-to-end message latency in a datacenter is down to a few microseconds, and since clock speeds for cores are not increasing, overhead in the software stack is one of the few available places for optimization.

### 5.4.2 Interrupts in L4

There's a useful correspondence between fast IPC mechanisms and interrupts. In a monolithic kernel like Unix, all the device drivers are inside the kernel and so no user space program ever needs to see an interrupt.

In a system with device drivers in user space like the L4 microkernels or Barrelyfish, however, a device interrupt has to be delivered to a user program as quickly as possible.

With a fast IPC mechanism already in place, the basic idea is to turn an interrupt into a message. Each device driver registers an IPC endpoint with the kernel when it wants to receive a specific interrupt vector.

When an interrupt is raised on the processor, the first-level Interrupt Request (IRQ) handler runs in the kernel and first masks the interrupt: since this interrupt will be handled ultimately by a user-space program which itself might be preempted, it's a good idea to make sure that the device does not raise another interrupt until the driver is finished.

Next, the kernel creates a message to be sent to the registered endpoint, makes the driver thread runnable, and hopefully dispatches it quickly. The driver then handles the message (presumably causing it to interact with the device hardware via I/O registers or shared data structures) and then replies to the message.

The kernel responds to this message reply by unmasking the interrupt ready for the next time. Thus a device driver has an "interrupt thread" which acts as a "server" for interrupts sent from the kernel.

### 5.4.3 Programming interface

L4 also has an interesting interface to message passing, which influenced the design of the Barrelyfish interface. The API defines a number (quite a large number) of *IPC message registers* or MRs. These are "virtual" registers, in that they form part of the programming model but not necessary part of the hardware architecture. However, they are considered part of a thread's execution state, and so they are "saved" and "restored" on a thread context switch. This model of MRs is then mapped onto the hardware in an architecture-specific fashion. IPC operations amount to copying data from the sender's MRs to the receiver's MRs.

On ARMv7-A, for example, the first 6 MRs are mapped to ARM processor registers and the remaining number are fields in the user-writable part of the thread control block. This means that IPC calls on L4 with small numbers of arguments are much, much faster than those with lots of arguments.

The precise number of MRs available in L4 is a system configuration parameter decided at build time. It is always at least 8, and never more than 64.

This scheme can efficiently transfer a number of machine words quickly, but how

these words are interpreted by both ends of the communication constitutes a network protocol, albeit on a single machine.

In this protocol, the first MR stores the *message tag*, which in a classical network protocol would be called a “header”. This header defines message size, and other metadata like how it is to be delivered (see above). The protocol views the rest of the payload as untyped words which are not normally interpreted by the kernel.



*Technical  
Details*

## 5.5 Lightweight message passing in Barreelfish

IPC in Barreelfish is strongly linked (as in systems like L4) by the architecture of the kernel, or CPU driver. It actually uses two very different mechanisms, one for communication between dispatchers on a single core, and one for communicating between dispatchers on different cores. This chapter we'll focus exclusively on the former, which is known as Lightweight Message Passing (LMP).

Since Barreelfish has a non-blocking kernel, LMP is also non-blocking. If a send can't get rid of its data immediately, it returns an error and the sending process has to retry later.

A receive operation is *also* always non-blocking, and so receive needs to be retried until something arrives. This is, on the face of it, highly inefficient. However, we will see that because of upcalls, it's not really a problem: LMP will always unblock (make runnable) a dispatcher it sends a message to, and a hint mechanism allows a newly-scheduled dispatcher to figure out quickly which messages it has been sent.

For the same reason, plus the fact that it uses upcalls for dispatch, communication is purely one-way (hence the name) as far as the kernel is concerned – there is no space in the CPU driver for keeping track of which RPC calls are in progress. RPC can, and is, implemented above LMP in user-space, but using many of the tricks of LRPC and L4 to make it fast.

Communication is always between two dispatchers in LMP, since the CPU driver doesn't really have a concept of threads.

As in LRPC (and unlike L4), binding is carried out in advance. A client binds to a server's *end point*, which is represented by a different type of capability (obtained by retyping the dispatcher control block's capability). Once a client possesses the appropriate end point capability, it can use this to invoke operations on the server.

Finally, LMP invocations can send capabilities as well as simply data.

We'll look in more detail at the programming interface next. You may find it helpful, however, to also look at the source code of libaos at the same time.



## 5.6 LMP programming interface

The low-level Barrelyfish library provides a very thin interface above capability invocations to implement LMP. It is, by design, not portable between machines (or, in fact, between different cores on the same machine!), and is more primitive than a programmer would like.

In most cases, regular Barrelyfish uses stub code generated by a compiler (`frounder`) to provide a nicer interface. However, in this chapter, we'll be using the raw interface.

While LMP is non-blocking at the CPU driver interface, user-level Barrelyfish threads can block on LMP bindings. The interface to this is based on `waitsets`, which are analogous to the sets passed to system calls like `select()` or `epoll_wait()` in Unix. LMP bindings can be added to or removed from a waitset. By default, all bindings held by a dispatcher are part of the *default waitset*.

Some of the most commonly used calls are shown below:

<code>lmp_chan_send( X )</code>	Send a <i>X</i> -word payload
<code>lmp_chan_recv()</code>	Receive a message (that's already there)
<code>lmp_chan_register_recv()</code>	Register a callback, to be notified when a message arrives on a channel.
<code>get_default_waitset()</code>	Return the default waitset for a domain

Here is a simple (actually, too simple) example of function which sends a (small) buffer of data over a LMP channel end point – essentially emulating network-style buffer passing over the register-based interface provided by LMP:

```

1 errval_t send_message(struct lmp_chan *c, uintptr_t *msgbuf,
2                         size_t msg_words, size_t *words_sent)
3 {
4     uintptr_t buf[LMP_MSG_LENGTH];
5     if (msg_words > LMP_MSG_LENGTH) {
6         msg_words = LMP_MSG_LENGTH;
7     }
8     memcpy(buf, msgbuf, msg_words*sizeof(uintptr_t));
9     memset(buf+msg_words, 0, (LMP_MSG_LENGTH-msg_words));
10    if (*words_sent) { *words_sent = msg_words; }
11    return lmp_chan_send9(c, LMP_SEND_FLAGS_DEFAULT, NULL_CAP,
12                          buf[0], buf[1], buf[2], buf[3], buf[4],
13                          buf[5], buf[6], buf[7], buf[8]);
14 }
```

All error checking has been omitted here for clarity, but you should not omit this kind of thing in your code!

The Receive example below is a bit more complex, as it illustrates the use of *callbacks* which the waitset provides to avoid having to keep polling the channel by repeatedly calling `lmp_chan_recv`. The receive-handler is deregistered each time it is called, so we need to re-register it each time:

```

1 errval_t recv_handler(void *arg)
2 {
3     struct lmp_chan *lc = arg;
4     struct lmp_recv_msg msg = LMP_RECV_MSG_INIT;
5     struct capref cap;
6     err = lmp_chan_recv(lc, &msg, &cap);
7     if (err_is_fail(err) && lmp_err_is_transient(err)) {
8         // reregister
9         Imp_chan_register(lc, get_default_waitset(),
10                         MKCLOSURE(recv_handler, arg));
11    }
12    debug_printf("msg_buflen_%zu\n", msg.buf.msglen);
13    debug_printf("msg->words[0]_=0x%x\n", msg.words[0]);
14    Imp_chan_register(lc, get_default_waitset(),
15                      MKCLOSURE(recv_handler, arg));
16 }
17 void some_func(void) {
18     // assumption: we have channel here
19     Imp_chan_register_recv(chan, get_default_waitset(),
20                           MKCLOSURE(recv_handler, chan));
21 }
```

It should be fairly clear by now that LMP, and (it turns out) most Barrelnfish programs are fundamentally event-driven. Each program usually has a main loop dispatching messages received on a given waitset, in a construction a bit like this:

```

1 int main_loop(struct waitset *ws)
2 {
3     // go into messaging main loop
4     while (true) {
5         err = event_dispatch(ws);
6         if (err_is_fail(err)) {
7             DEBUG_ERR(err, "in_main_event_dispatch_loop");
8             return EXIT_FAILURE;
9         }
10    }
11    return EXIT_SUCCESS;
12 }
13 int main(int argc, char *argv[])
14 {
15     // do initialization
16     // ...
17     // run messaging loop on default waitset
18     return main_loop(get_default_waitset());
19 }
```

## 5.7 Stack ripping

Event-driven programming, as you will probably discover, is not without its problems. The need to structure your code as a set of event handlers, rather than a single inline imperative program, makes it harder to both write and read. The style required is called “stack ripping”, and is well-described in the paper that introduced the term [3]. The core logic of a program ends up being split across a chain of callback functions. This is, arguably, a more faithful representation of what is *actually happening*, but is much further away from what a programming *actually wants to express* than, say, RPC.

Here’s an imaginary example, of calling a name server (see chapter 12) to look up a string, written in stack-ripped style:

```

1 void Lookup(NSChannel_t *c, char *name) {
2     OnRecvLookupResponse(c, &ResponseHandler);
3     // Store state needed by send handler
4     c->st = name;
5     OnSend(c, &SendHandler);
6 }
7 void ResponseHandler(NSChannel_t *c, int addr) {
8     printf("Got_response_%d\n", addr);
9 }
10 void SendHandler(NSChannel_t *c) {
11     if (OnSendLookupRequest(c, (char *)(c->st)) ==
12         BUSY) {
13         OnSend(c, &SendHandler);
14     }
15 }
```



*Background*

Fortunately, we can do better. There have been several proposals for turning event-driven execution into something resembling single-stack code, by using constructs a bit similar to coroutines (e.g. [68]).

Barrelfish has a similar facility known as Asynchronous C (AC) in publications, and Tim Harris’ C (THC) in the source code [55]. THC extends regular C with three new constructs:

- `async statement`
- `do { block } finish`
- `cancel`

These can be compiler extensions, but can also be provided (at some cost in elegance) by C preprocessor macros. Their goal is to make message-passing code scalable and readable.

Here’s an example where two lookups are fired off in parallel, and we wait for both responses:

```

1 // Caution: functions ending in AC may block
2 void LookupAC(NSChannel_t *c, char *name) {
3     int addr;
4     SendLookupRequestAC(c, name);
5     RecvLookupResponseAC(c, &addr);
6     printf("Got_response_%d\n", addr);
7 }
8 void TwinLookupAC(NSChannel_t *c1, NSChannel_t *c2, char *name) {
9     do {
10         async LookupAC(c1, name); // S1
11         async LookupAC(c2, name); // S2
12     } finish;
13     printf("Got_both_responses\n"); // S3
14 }
```

Crucially, there is *no multi-threading* in these examples – it's all on one thread. The extensions simply identify opportunities where multiple messages can be issued concurrently.



#### Technical Details

## 5.8 The milestone: overview

Your goal is to demonstrate passing small messages between processes using LMP.

First, you create an endpoint capability by *retyping* the capability from the dispatcher capability. There is a function to do this in the CPU driver, and a corresponding system call to do this in user space. However, after retyping the domain dispatcher capability to an endpoint, you are not quite finished. Barrelyfish doesn't let you use that endpoint for sending messages because each endpoint needs to have a buffer associated with it. You can use the *mint* operation for this.

Once you've got a proper endpoint capability in your own domain, you can think of it as like a non-blocking server socket: when a message arrives for the endpoint, an upcall is made to your domain with additional information indicating which endpoint the message is for.

Conversely, to send a message to a domain you need to have acquired a copy of the receiver's endpoint capability. This can then be "invoked" (again, there is a system call for this) which sends a message containing a small set of arguments in registers.

Like all networking systems, there is a bootstrapping problem: how does one domain get hold of an endpoint capability for another domain? To start with, we'll fix this by installing the right capabilities in each domain when they are getting spawned. You should modify the spawning code to pass a newly created endpoint capability at a well-known location in the new domain's CSpace.

Now, you should be able, at a very low level, to send a message from the second domain to `init` by invoking, in the second domain, the endpoint capability which is a copy of the one retyped from `init`'s DCB. Make sure that `init` also has the corresponding endpoint capability of the other domain. Later you may use the capability transfer functionality to request and distribute endpoint capabilities.

The best way to understand both the API for sending a message, and the in-kernel implementation that performs the message send, is to look at the function `handle_invoke` in the file `syscall.c` in the subdirectory of `kernel/arch/` corresponding to the architecture ARMv8-A. You'll see that a number of flags can be specified when sending a message, which give hints as to what domain to run next. The rest of the code to implement LMP is in `kernel/dispatch.c`.

Note that sending a message will not automatically succeed – for example, if the receiving domain is not in a position to process the message, you will get a negative acknowledgment back from the system call. In this case, the best approach for now is to yield the processor to allow the other domain to run, and try again later.

## 5.9 Getting prepared

This milestone will use a new domain (`/usr/memeater`). Make sure `memeater` is added to the list of modules (`/platforms/Hakefile`) to be built and also to the `menu.lst` you are using.

Recall, we do not need any special options passed to the linker. If you compare `memeater`'s Hakefile with that of `init`, you will see that `memeater` has no option `-e _start_init`. This special option is required for `init` since it has a slightly different initialization path that stops before the message passing infrastructure is initialized.

Run Hake again, make `clean`, and do a clean build. This should give you a working image to start with.



*Project  
Instructions*

## 5.10 Implementing your channels

Dealing with the raw, `syscall` interface of LMP is a bit tedious, so you want to provide appropriate abstractions. In the end, you will need to implement the specified RPC interface ([Listing 5.1](#)). We will provide you with the low-level abstractions such as `lmp_endpoint` and `lmp_chan`. Your task will be to provide functionality to distinguish different message types, marshaling unmarshaling of messages and RPC semantics.



*Project  
Instructions*

### 5.10.1 Endpoints and Channels

We already provide the code to manage raw LMP channels in `lib-aos`.

Remember the bootstrap problem we described earlier. The first goal is that a newly-spawned process is able to communicate with `init` to obtain more memory (for example). To set up the required channel you need a LMP endpoint capability for each side of the channel. You can use the function `endpoint_create` which takes a dispatcher capability and will create a new endpoint capability. The LMP channel record `struct lmp_chan` has fields to store the local and remote endpoint, and you will need to set the corresponding capabilities in this structure. In the child, the remote endpoint was passed by the parent at a well known location in the child's CSpace. In the parent, the remote endpoint is received from the child.

You can set up your `struct lmp_chan` structures for both ends of the connection. You'll have to make sure that on `init`'s end, the local endpoint corresponds to the endpoint we've been using for the previous step.

### 5.10.2 Sending and Receiving Messages and Capabilities

Once you have set up your LMP channels, you can start sending or receiving messages using `lmp_chan_recv` and `lmp_chan_send`.

On the receiving side, you will get a pointer to a message structure containing the message payload. You can also receive capabilities in a message, but you will need to provide empty slots in your CSpace in order to receive them, which in turn requires you to allocate these slots in advance. There is functionality in the user-level LMP machinery to set an empty slot for receiving a capability. This will later become important when you want to request more memory.

On the sending side, you can simply call `lmp_chan_send()` (or one of its variants in `include/arch/arm/aos/lmp_chan_arch.h`) with the appropriate arguments. Your channel abstraction will need to take care of marshaling the payload into these arguments.

Note that receive can fail if there is no message to be received, and/or that send can fail if the other side has not called receive and the message cannot be buffered. One approach would be to busy-wait until you can send or receive, but on a single core this is not likely to allow the other side to catch up. Next we will see what you can do to avoid this.

### 5.10.3 Event handling

At this point, you have set up the LMP channel. However, there is still work to do before you can use the channel, in particular handling *events*. Events in combination with message passing are a central element in Barreelfish. Services wait until they are notified by means of an event that a message can be received, they perform the actions related to the request, and finally (depending on the request) send a reply back.

A key element of Barreelfish's event handling is the *waitset*, which forms the connection between a *thread* and an *event*. The code for handling an event is represented

as a closure (a function pointer and arguments). In LMP, these events are raised by message channels in response to activity (such as a message arriving and becoming available to receive). Events are added to waitsets, and threads wait on waitsets before dispatching corresponding handlers when the event occurs.

Each LMP channel therefore should have a waitset associated with its receive handler, and a thread in the domain *dispatching* events on that waitset, for example by calling `event_dispatch` on the waitset in an endless loop. For this reason, you need to provide a waitset when you set up your channel abstraction. You can use a single waitset for all channels (see `get_default_waitset()`) or, alternatively use separate waitsets for different RPC channels.

You can register for receive events on a LMP channel using the provided function `lmp_chan_register_recv()` (and the analogous call can be used for the sending case). You will need to supply as arguments the LMP channel, the waitset you want to associate the LMP channel with, and an event closure to be called when there are new events on the LMP channel. The event closure will be your receive-handler and an argument you can use to refer to state.

#### 5.10.4 Your channel abstractions

In the end, you have to implement the RPC interface stated below in [Listing 5.1](#), but there are multiple ways to do this. For example, you could implement the RPC interface on top of the low-level LMP channels, or perhaps above some other, intermediate abstraction. As a heads-up, later on (in [chapter 8](#)) you will need to write a cross-core RPC implementation over shared memory, which uses a very different mechanism to LMP. Keep this in mind when designing and implementing your solution.

You will need to think about what state you need to track across invocations. It's a good idea to store the channel to `init` at a well-known location in your domain's CSpace. Furthermore, you will need to define a protocol and message format, as you will need to be able to handle multiple message types. In particular, you will need to implement functionality to send numbers, request new RAM capabilities and also handle strings. You can limit the maximum size of the string or support sending variable-sized strings (see the extra challenge below).

Other than this, you have a great deal of implementation freedom in this task, and there are plenty of perfectly good approaches. You simply have to conform to the RPC interface (see `include-aos-aos_rpc.h` for the full interface definition) and document and motivate your implementation. The code excerpt below ([Listing 5.1](#)) reproduces only the client-side RPC functions for this milestone; you will have to implement the remaining RPCs given in `include-aos-aos_rpc.h` in later milestones.

Listing 5.1: RPC Interface

```
1 struct aos_rpc {  
2     // TODO: add state for your RPC implementation
```

```

3 };
4 /*
5  * RPCs on the init channel
6  */
7 // obtain channels to the various services
8 struct aos_rpc *aos_rpc_get_init_channel(void);
9
10 errval_t aos_rpc_send_number(struct aos_rpc *chan, uintptr_t val);
11 errval_t aos_rpc_send_string(struct aos_rpc *chan, const char *string);
12
13 /*
14  * RPCs on the memory channel
15 */
16 struct aos_rpc *aos_rpc_get_memory_channel(void);
17
18 errval_t aos_rpc_get_ram_cap(struct aos_rpc *chan, size_t bytes,
19                             size_t align, struct capref *retcap,
20                             size_t *ret_bytes);
21
22 /*
23  * RPCs on the serial channel
24 */
25 struct aos_rpc *aos_rpc_get_serial_channel(void);
26
27 errval_t aos_rpc_serial_putchar(struct aos_rpc *chan, char c);
28 errval_t aos_rpc_serial_getchar(struct aos_rpc *chan, char *retc);
29
30 /*
31  * RPCs on the process channel
32 */
33 struct aos_rpc *aos_rpc_get_process_channel(void);
34
35 errval_t aos_rpc_process_spawn(struct aos_rpc *chan, char *cmdline,
36                                coreid_t core, domainid_t *newpid);
37 errval_t aos_rpc_process_get_name(struct aos_rpc *chan,
38                                   domainid_t pid, char **name);
39 errval_t aos_rpc_process_get_all_pids(struct aos_rpc *chan,
40                                       domainid_t **pids,
41                                       size_t *pid_count);
42
43 // initialization
44 errval_t aos_rpc_init(struct aos_rpc *rpc);
45
46
47
48

```

The interface consists of three different types of functions.

**Initialization:** The `aos_rpc_init()` will initialize the state for an RPC and makes

it available to use afterwards.

**Channel:** The `aos_rpc_get_XXX_channel(void)` functions return an already setup RPC channel to the respective service. It's also possible to return the very same RPC channel for all services.

**RPC:** The remaining functions are actual RPCs that will send a message and wait for a reply on the channel.

As you can see, there are four different channels: *init*, *memory*, *serial*, and *process*. You are free to implement those logical channels over independent LMP channels, or you can use a single LMP channel for all RPCs. No matter what approach you take, you must ensure that the corresponding RPCs are valid on the returned RPC channel from a call to the respective `aos_rpc_get_XXX_channel(void)` function. See the following example:

```
1 struct aos_rpc *memchan = aos_rpc_get_memory_channel();
2 err = aos_rpc_get_ram_cap(memchan, 4096, 4096, &ramcap, &retbytes);
```

The idea behind this RPC interface is that, quite often, operations like requesting a RAM capability from some other domain (most likely a memory server – c.f. the next step) are actually comprised of sending a message and waiting for the reply to that message. In the end you should be able to get a new RAM cap as in the example above.

The purpose of the `struct aos_rpc` is to keep state for your RPC channel (e.g. the underlying LMP channel, the currently pending replies). You may change the function signature of `aos_rpc_init()` to accommodate setting up a RPC channel's state with whatever your implementation needs.

Note: If you change the function signatures in the interface (apart from `aos_rpc_init()` as mentioned) you'll need to give a good explanation why this is necessary and document the changes as we will write test programs using that interface. You are free to add *additional* RPC functions, but you should make sure to implement the required ones.

Once your channels work, make sure that you setup the RPC client to init in `lib-aos/init.c`, in the function `barrelfish_init_onthread()`.

Once you have a fully initialized RPC client for `init`, you can use the function `set_init_rpc()` to store it in your application's "core state". This allows you to then use the RPC client for `init` by just getting it by calling `get_init_rpc()`. See `lib-aos/domain.c` to see how these two functions work.

For each of the RPCs you are supposed to implement we also added a function call that you should call on the receive/server side (receive handler) of your RPC channel. This is in order to grade your solution. The function calls are shown below.

```
1 /* Stubs for grading RPC implementation
2  * These calls should be called on the receiver side of the RPC defined in
3  * lib-aos/aos_rpc.c
```

```

4  */
5 void grading_rpc_handle_number(uintptr_t val);
6 void grading_rpc_handler_string(const char* string);
7 void grading_rpc_handler_serial_getchar(void);
8 void grading_rpc_handler_serial_putchar(char c);
9 void grading_rpc_handler_ram_cap(size_t bytes, size_t alignment);
10 void grading_rpc_handler_process_spawn(char* name, coreid_t core);
11 void grading_rpc_handler_process_get_name(domainid_t pid);
12 void grading_rpc_handler_process_get_all_pids(void);

```



*Extra  
Challenge*

## 5.11 Passing large messages

LMP on Barreelfish can only transfer a few machine words and an optional capability. Of course, messages often need to be larger than this. There are several approaches to this problem.

First, you can write a small piece of software (a “stub”) which breaks a large message into several smaller ones, and another stub on the receiving side which assembles the pieces before delivering the larger message.

Second, you can use a special area of memory on the sender and receiver, and have the kernel copy more data between these buffers during the call. This is tricky, because you have to make sure no other thread is using the buffers when a message is sent.

Third, you can allocate a new frame capability large enough to hold the message contents. Map it in the sender, copy the message contents into it, pass it to the receiver, and clean up. The receiver then maps it into its address space and reads the content. This method is quite heavy weight as

Fourth, you can create a special area of shared memory between two domains, and make sure both domains have a capability to it. They can then map this into their own address spaces, and use it to pass messages. This is a lot of work to set up (in particular, the exchange of capabilities for the memory) and tear down when done (likewise), but is the best approach when messages are very large, and the only way to go between cores (as we’ll see later in the course).

## 5.12 A memory server

You have now got all the basic features required for IPC: bootstrapping communication, passing data, and passing references to further communication endpoints.

The next step is to do something useful with this: implement a memory server which can allocate regions of physical memory to other domains, and hand over capabilities to those domains.

You can implement the server in any way you like. One simple way is to run it in the `init` domain as you already have the facilities for managing memory in `init`. Other domains can use their existing communication channels to `init` to request and receive memory.

More elegant, but more work, is to implement the memory server as a separate domain (Barrelfish itself employs a variation of this technique). This also requires you to provide a way for domains to request an endpoint capability for the memory server from some other domain which they can already talk to (such as `init`); essentially, this involves implementing a simple name server or to pass the endpoint capability at a well known location in the domain's capability space upon spawn.

Regardless of which design decision you chose, make sure you return the right channel when the function `aos_rpc_get_memory_channel` is called.

You should also think about the operations that the server supports and the invariants. For example, clearly you need to make sure that each client receives capabilities to disjoint areas of memory in response to their requests. You might also want to limit the quantity of memory each client receives, to prevent a client from grabbing all the available physical memory. Important, state the policies you implement in the documentation.

As a final step, allow new domains to perform memory allocation on demand using your new memory server and the `aos_rpc_get_ram_cap` RPC call. After you obtained a new frame capability, you can use your paging infrastructure from the previous milestone to map it and access the memory on it. Note, you will obtain a RAM capability which needs to be retyped. Barrelfish already has wrapper functions that handles this: `frame_alloc`. You will need to reset the RAM allocator to use your RPC. During the initialization of `libaos`, you can reset the RAM allocator by a call to:

`lib-aos/init.c`

```
1 ram_alloc_set(NULL);
```

and provide the corresponding implementation in

`lib-aos/ram_alloc.c`

```
1 static errval_t ram_alloc_remote(struct capref *ret,
2                                 size_t size, size_t alignment)
```



As a hint, receiving a capability always uses a slot in your CSpace. If you run out of slots, you will need to get new memory to create a new CNode to hold new capabilities. You may want to make sure that you always have enough free slots available.



*Project  
Instructions*

## 5.13 Spawning Domains

Currently, you should be able to spawn new domains within `init`. However, also other domains need to have the ability to spawn programs. A central use case of this is a shell accepting user input and spawning programs as requested.

You should implement the RPC for process management, i.e., so you can send a request to spawn new processes. If you have implemented the extra challenge in the previous milestone, you could try to get a more complete process management support going.

One thing you can try is sending a *bye* message to `init` when a process terminates.



*Project  
Instructions*

## 5.14 A terminal driver

Currently, you perform a syscall (`sys_print`) to the CPU driver when you do a `printf`. You can now instead use your communication framework to call `init` or another domain as a “terminal” driver. For this purpose you will need a way to send characters (or strings) between the domains. You can use your `send_string` RPC or send a single message. In order to use your own terminal read and write functions you need to set the libc function pointers accordingly. Change the lines in `barrelfish_libc_glue_init` to point to your new read/write functions. For this milestone you need to implement the terminal write functionality.

lib-aos/init.c

```

1 void barrelfish_libc_glue_init(void)
2 {
3     /* optional */
4     _libc_terminal_read_func = aos_terminal_read;
5     _libc_terminal_write_func = aos_terminal_write;
6     /* ... */
7 }
```



*Extra  
Challenge*

## 5.15 Optimization

Traditionally, operating systems (in particular microkernels) have tried to make communication between two processes on the same core as fast as possible.

You can measure how long it takes between sending a message and receiving it on the other side using the performance cycle counter on the processor.

Our implementation in Barreelfish isn't bad, but it's almost certainly not as fast as it could be. See if you can make it faster (and demonstrate this), without removing any of the functionality it currently offers.



#### Commentary

## 5.16 How to do well

As with all the milestones in this course, it's not enough to show that messages can be sent – the facility has got to *work* as part of an OS. You're going to be relying on this extensively later in the course, so it's really a very good idea to get it working well now.

Make sure that you really understand (and can explain) what happens when a LMP message is sent from one domain to another. There's quite a bit of detail in here, some of which you will have implemented, and some of which was in the code we provided (which you should have thoroughly grokked by now).

You also need to understand how a channel is created; most people (me included) tend to focus on the business of sending and receiving messages, but the process by which a channel is established, and indeed torn down when it is no longer needed, is often more complex.

LMP is only available for domains running on the same core. Later on, you will also need to deal with cross-core communication based on User-level Remote Procedure Call (URPC) ([chapter 8](#)). A well-designed RPC infrastructure will facilitate future milestones.

Finally, while this is all happening within a single operating system, you should have realized by now that you have implemented a network *protocol*, with all that that entails. Can you explain it in these terms?

## 5.17 Milestone 3 Summary

Milestone tasks:

- Implement the RPC functions and the corresponding server implementations.

Extra challenges:

- Handle large messages.
- Demonstrate optimizations of the LMP code.

Assessment:

- That you can correctly invoke an endpoint capability and receive a message on the same capability in another domain.



#### Project Instructions

- Demonstrate that your RPC implementations are working.
- Demonstrate that you can spawn new domains upon request.
- Demonstrate and explain a memory server process allocating RAM caps among multiple domains.
- Demonstrate and explain how your terminal service works.

Try to make sure your code is bug-free. We'll expect you to demonstrate a system that can handle an arbitrary amount of domains talking to each other and each of those domains should be able to talk to the memory server when it needs more physical memory.

# Chapter 6

## Page fault handling

In this milestone, you will implement a page fault handler as the core component for *self-paging* user processes. You will use the functionality you implemented in the previous milestones to obtain and manage RAM capability to serve your application's memory needs, and extend your virtual memory management code.

### 6.1 A different view of virtual memory



Commentary

Apart from learning about fault handling and paging, this milestone exposes you to what may be a very different way of thinking about paging than what you're used to from the Unix world.

You won't be building complete demand paging to disk (or some other storage device at this stage), but you will have to handle page faults. The goal for this week is to perform *lazy allocation* of the program heap (the memory allocated by `malloc`): instead of making sure that all the physical memory for the heap is there at the start of the program, you will simply reserve the *virtual* memory, and then fill it on demand with physical memory in response to page faults in that area of the address space.

#### The work consists of:

- Implementing an exception handler
- Heap management with lazy allocation
- Implementing the self-paging mechanism
- Handling page-faults in user space

We'll first look at the classic Unix view of virtual memory, and then discuss the history of an alternative approach based on user-managed virtual address spaces.

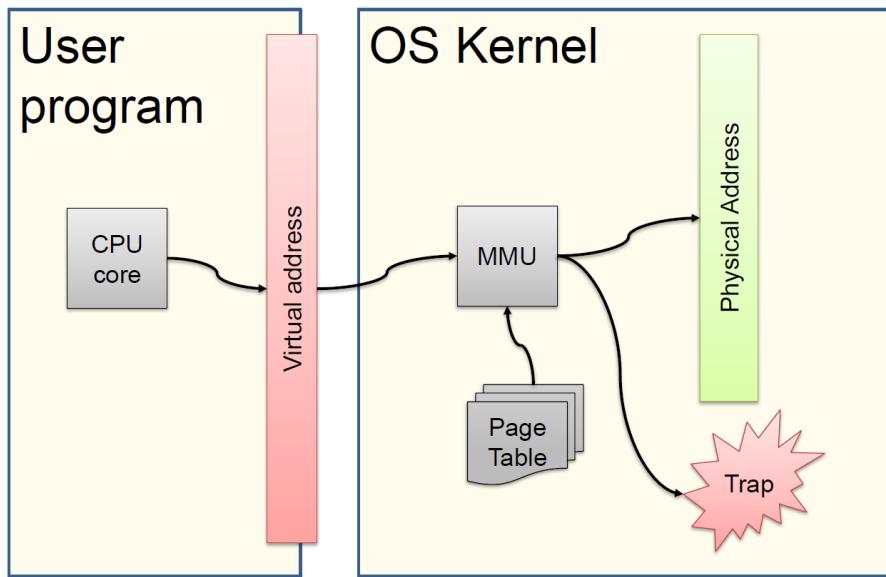


Figure 6.1: The Unix API to virtual memory

After that, we'll go into the details of how the latter is done in Barreelfish, and what you'll be implementing.



*Background*

## 6.2 The classic Unix memory API

To start with the familiar, figure 6.1 shows the classic way that virtual and physical memory are *mostly* presented to user programs in Unix, and indeed most other mainstream operating systems.

In its “pure” form, the interface works as follows:

1. User programs work entirely with virtual address spaces.
2. A user process sees its own complete (32-bit or 64-bit) virtual address space which appears to be full of DRAM.
3. Actual physical memory to back this virtual address space is allocated on demand, and pages may be transparently paged out to storage if there's not enough memory in the system.
4. Page faults are not visible to user programs

This interface is very simple and easy to use, and for many programmers it appears to be all that is needed. However, it's not sufficient to write performance programs in many cases, and the flip side of its simplicity is that it is highly opaque

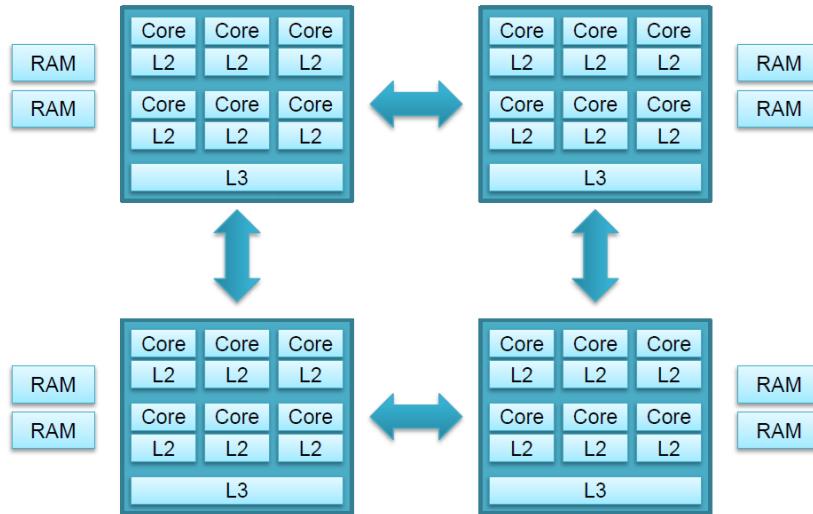


Figure 6.2: A typical modern 4-socket NUMA machine

and inflexible. For this reason, over the years most versions of Unix have punched an ever-increasing number of “holes” in the clean abstraction of a virtual address space to make it more and less “in the way”.

The first problem is that **physical addresses matter**. On a Non-Uniform Memory Access (NUMA) machine (e.g. Figure 6.2), the physical address of a page of memory affects the latency with which it can be accessed depending on which core (or last-level cache) in the system is reading or writing it.

Conversely, in a machine with multiple memory controllers allocating all physical memory from DRAM attached to a single controller limits the bandwidth available to main memory from software, and ideally physical memory is allocated from all controllers to load-balance across NUMA nodes.

Any programs performing high-performance I/O, including Remote Direct Memory Access (RDMA), need to *pin* virtual addresses so that the physical memory backing them is always present and does not change, until the I/O operation is complete.

Increasingly, memory is not only distributed around the system, but has different types: regular DRAM, graphics memory such as GDDR, high-bandwidth memory close to a core, scratchpad memory for accelerating certain operations, etc. The physical address determines what kind of memory you access when issuing a virtual load or store.

In a cache which is not fully associative (and few are), the physical address also determines which set is used to cache the contents of memory. If you are using *cache*

*coloring* techniques to reserve parts of the cache for certain processes in order to provide performance isolation, you care about the physical address of memory.

Some devices, including GPUs, or other accelerators like Xeon Phi, but potentially any DMA-capable peripheral, may only be able to access certain areas of physical memory. If you want to pass data to such devices in shared memory, you need to be careful where it is.

The second problem is that **translations matter**. As memories get larger and larger, small pages (4KiB seems pretty common) become quite inefficient – they require very large page tables and increase the miss rate of the TLB. Most MMUs provide “superpages”, large page mappings encoded by terminating the page table walker early, and these can improve performance of many workloads. However, superpages are not a silver bullet: they actually slow down quite a lot of significant workloads as well. Demand paging is one example: the latency introduced by paging out 1GB superpages is pretty significant.

Even with a constant page size, translations matter. For example, they can be used to provide *intra-application* protection.

The third problem is that **traps matter** to user programs as well. A classic paper by Appel and Li from 1991 [10] detailed a number of ways in which an application program, and in particular a garbage-collector in a language runtime, can benefit from being able to cause, and catch, page faults on particular virtual addresses, independent of actual demand paging.

Systems like Linux have evolved over the years to “punch holes” in the clean Unix abstraction of a pure virtual address space. For example, ranges of virtual addresses can be “pinned” to prevent them from being paged out. You can create regions of virtual memory and then change protection bits using `mmap()`, `mprotect()`, and the like. Page and segmentation faults can be turned into Unix signals and directed back to the faulting program. Linux has at least three different ways of supporting superpages: via a file system, explicitly allocating them from a fixed pool determined at boot time, or transparently creating them from suitable-looking regions of virtual mappings. All these interfaces operate *implicitly* on real memory via virtual addresses.



### Commentary

## 6.3 Stepping back a bit

If we think about it, a MMU performs two tasks, translation and protection, and serves two masters, the process and the OS.

The process is mostly interested in translation. Translation (more accurately, both translation and other forms of interposition on accesses) has many potential uses for a user process:

**Demand paging** is the original use for virtual memory, and allows a program to access more virtual address space than the amount of physical RAM available

to it on the machine.

**Relocation** is often overlooked, but is a critical part of how modern OSes work: a program can be loaded anywhere into physical memory even though it was compiled and linked to run at a particular virtual address.

**Interposition** is, as the Appel and Li paper points out, useful for a garbage collector to figure out if a page has been touched or not. Such functionality has also been used recently by databases to detect conflicts in transactions [48]. User-level threads packages have used interposition to detect stack overflows.

**Replication** using copy-on-write techniques has also been successfully used in programs to avoid pre-copying large quantities of data until (or if) it is needed.

The Unix interface was originally designed only for demand paging, which is arguably the *least* important modern use of an MMU. Who really pages these days? It's not a useful feature on, e.g., a mobile phone. As storage devices get faster and faster, it may come back as useful feature, but until recently when SSDs became the norm, if your laptop started seriously paging to a hard disk often the easiest thing to do was reboot it.

In addition, processes are sometimes also interested in protection: sandboxing of user code, such as plugins or code embedded in web pages, or protecting regions of the stack to guard against exploits of software bugs. This is protection *within* a process.

The OS is mostly interested in protection *between processes*. As long as processes can't access each other's memory, for correctness an OS kernel doesn't need to do much more. Most of the complexity of the kernel implementation of virtual memory on Unix is about creating a *convenient* single abstraction for the user rather than providing performance or safe isolation between untrusting processes.

## 6.4 User-managed virtual memory

As you probably realized in [chapter 3](#) Barreelfish, like some other systems, is very different from the *virtual address* orientation of Unix. Instead, *physical* resources are referenced directly using capabilities. As a result, the virtual memory system looks rather different, even if it can be made to emulate a pure Unix model if required. We'll be working with this alternative design in detail in this chapter.

Many OS designs have adopted an approach to virtual memory that is very different to the Unix model. Some microkernels like Chorus [1], Mach [92], and Spring [64] have used server processes to back regions of virtual memory in other processes, through what are sometimes called "memory objects" – objects that service page faults for some part of an address space. Different classes of memory objects can be used to implement demand-filled physical memory, disk paging, copy-on-write semantics, etc.



Commentary

However, this is a bit different from a user process handling *its own page faults* in a library, as happens in Barreelfish. This has also been done before, notably in the Cache Kernel [56], Exokernel [44, 42], and Nemesis [52].

In these systems, the Unix-style model is completely inverted. Recall that a modern Unix system starts with the illusion of a uniform virtual address space backed by unlimited RAM, and then pokes holes in this abstraction to allow virtual regions to be pinned in memory, allocated from particular NUMA regions, allocated from particular page pools, protected in particular ways, etc.

By contrast, in a *self-paging* system the process constructs its own set of virtual address mappings by explicitly acquiring physical resources (in effect, regions of physical address space corresponding to different kinds of memory), and then constructing its own page tables to program the MMU to arrange these resources in virtual memory. Finally, it handles its own page faults by rearranging these mappings.

On top of this rather more exposed interface, library code can construct Unix-like abstractions if required. Typically, when the process starts up it is provided with a limited conventional virtual address space to get started.

Here, the goal is rather different from Unix. Instead of the kernel providing as simple and easy-to-use a model of virtual memory as it can, in self-paging systems the aim is for the kernel to allow a process to make most efficient, flexible, and creative use of the MMU as can be safely allowed.

This is not to say that ease-of-use is not important, but to recognize that a nice simple interface can be better provided in a user-space *library* than in the kernel. This philosophy is at the heart of exokernel-based OS designs [43, 45].



Technical  
Details

## 6.5 How does it all work?

If we are going to allow a user program to create and modify its own page tables, we need to make sure that nothing bad can happen. We can express this rather more precisely in this safety property:

**Safety:** A process cannot create a page table which grants it access to physical addresses for which it does not hold authorization.

This implies that an implementation has to have, firstly, a secure way of authorizing access to physical address ranges, and secondly, a way to check any page table for correctness before it is installed in the MMU (and while it is active).

All the self-paging systems mentioned above have some kind of “secure handle” approach to dealing with physical memory resource, which all boil down to capabilities. Barreelfish simply refers to these as capabilities, as we have seen in [chapter 3](#).

You’ve already implemented a memory allocator, but a full Barreelfish system can have potentially many different allocators: for example, one for each NUMA node

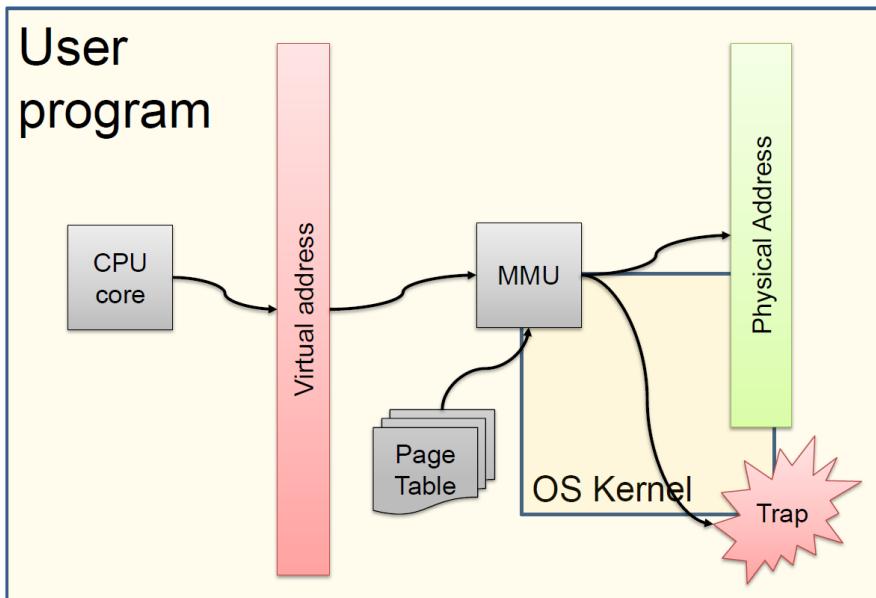


Figure 6.3: The Barreelfish API to virtual memory

or DRAM controller in the system, or one for “special” areas of memory (such as the low 4GB on a 64-bit x86 machine). Each allocates memory using capabilities, and so applications can explicitly acquire memory resources of particular types. Capabilities can also be revoked, which means memory can be reclaimed from uncooperative processes (we don’t ask you to implement that in this course).

When it comes to ensuring that only valid page tables can be constructed and used, there is an additional, desirable property that we would like our memory system to have, in order to be able to exploit any particular MMU hardware features that might be available – if we are going to remove the virtual address space abstraction and (safely) expose the page table, we might as well go the whole way and allow users to employ it any way they want:

**Completeness:** A process should be able to construct any safe page table the hardware allows, subject to resource limits.

This goal appears to be unique to Barreelfish, though possibly because prior systems were only targeting a single MMU design.

Finally, we also need to be able to handle page faults in the application. This is going to need some kind of exception mechanism - we can handle a page fault on the thread that took the fault, since it may not have access to its own code, stack, or heap (depending on the fault).

Instead, we want some kind of upcall mechanism. This can actually be done in Unix

with signals – catching SEGV in a process.

In Barrelyfish, as you know from [chapter 4](#), there is already an upcall mechanism that is fairly fundamental to how processes and user threads are implemented. It makes perfect sense, therefore, to notify a user process that it has taken a page fault by means of the same upcall that dispatches it next time we exit the kernel, and this is exactly what happens. There is actually a separate entry point in the user-space DCB for page faults, which could be used to handle page faults within the user-level thread scheduler, but right now we don't use this; it's an idea taken from K42 [[67](#)].

The corresponding Barrelyfish version of [Figure 6.1](#) is [Figure 6.3](#). The kernel or CPU driver is involved in anything that might be “unsafe” (actually writing PTEs, loading the page table base registers, catching the initial hardware exceptions), but as much as possible is pushed into the user library.



*Project  
Instructions*

## 6.6 Implement an Exception Handler

The first part of this milestone is very simple. To handle page faults you need a way to run custom code when an exception occurs. The first task is simply to make sure that you can catch a page fault and react to it – even if you don't actually do anything useful at this stage.

We provide a way to direct a page fault on a thread received by the dispatcher to a user-specified function: You can set an exception handler for a thread using the function `thread_set_exception_handler` in the `threads` package in `libaos`.

As a first step, we suggest you set this to a function which prints some information about the exception to the console and then stops. You can then test it by triggering a page fault: just invent an address which is currently not backed by memory (for example, somewhere a way above the top of the current heap), cast this to a `char *` and try to read it.

If this works, great – all we need to do now is make this function do something more useful like servicing the page fault and resuming the thread. First, however, we're going to need more infrastructure to keep track of where things are in the address space.

At this point, you might also find it helpful to refer back to the description of the MMU hardware in [Section 3.12](#).



*Project  
Instructions*

## 6.7 Design the Address Space Layout

Before we start creating and modifying page tables, it's important to think about how to represent the address space at user level, so that your paging code knows which regions are occupied, which are free and also which regions actually belong to the heap.

If you think of a page table as a data structure which maps virtual addresses to physical addresses using hardware, we're going to need to do the same translation in software, plus the whole thing in reverse: there are many cases where we need to know where a given physical page is mapped in the virtual address space.

Apart from any other reason, we're not interested in successfully handling page faults at any address. For the moment, it's just about the heap, which we'll be filling on demand by allocating more physical memory. You should already have the functionality to ask for a free region or to map memory at a specified address.

We won't give you much of a template. You can do this any way you like (a list, a tree, a hash table, ...) but keep in mind which operations need to be efficient at runtime, and be prepared to explain your choice in the marking session.

This task of the assignment, then, is to (re)design<sup>1</sup> the data structures in user space which keep track of all the memory mappings your process currently has set up. You should put all the state of your implementation in the `struct paging_state` of which you'll find an instance called `current` setup as a global variable in the file `lib-aos/paging.c`.

---

<sup>1</sup>You may already have designed a set of data structures that is adequate to handle the full set of requests introduced by this milestone, depending on the amount of effort you spent on the paging code in previous milestones



*Technical  
Details*

## 6.8 Programming the MMU

Recall, [section 3.13](#) on how to construct safe page tables. Once you've constructed a page table, how do you actually use it? A second capability operation:

```
switch( cap )
```

– actually installs a new page table in the MMU. More precisely, the argument `cap` *must* be a capability whose type represents a top-level page table native to the processor architecture that the code is executing on – thus, on an ARMv7-A processor, it must be of type `VNode_ARM_11`, on ARMv8-A it must be `VNode_AARCH64_10`, and on 64-bit x86 it must be `VNode_x86_64_pml4`.

This records the new page table in the DCB, loads the value into the MMU's page table base register, and carries out the necessary cache and TLB flushes.

Naturally, this is a somewhat hairy operation – just as with turning on an MMU in kernel initialization code, it's important to make sure that the next instruction is fetched from a reasonable address given that the mappings have all changed. However, note that you can't crash the OS this way: all you can do is crash your own program if you get it wrong. Note also that you can't install a new page table which gives you access to anything you don't already hold a capability for: safety is preserved.

However, as well as simply enabling self-paging, this ability to switch address spaces has other uses, particularly when trying to address very large arrays of data: it turns out that physical address spaces, even on 64 bit machines, are larger than virtual address spaces and you can actually start to run out of bits in the latter. A recent paper [\[39\]](#) proposed this mechanism for this and several other use-cases, and showed how to implement it in BSD, Linux, and Barrelyfish. The implementations in Unix-like systems are quite complex due to the need to translate extensively between virtual and physical address spaces; in Barrelyfish it's a handful of lines of code.



*Project  
Instructions*

## 6.9 Implement Page Fault Handling

You should now be able to implement full page fault handling in your exception handler, by calling the appropriate functions of your paging implementation, e.g. `paging_map_fixed_attr()` to install a newly acquired page of RAM at the faulting address.

While you're about it, you might also think about detecting NULL pointer dereferences and disallowing any mapping outside the range(s) that you defined as valid for heap, stack, etc. Moreover, consider adding a "guard" page to the process' stack – this is a protected page which will generate a fault if the stack space is exceeded. All these measures make debugging overflows or bad pointers a whole lot more pleasant, since you can print out both the faulting address and the program counter where it happened.

The key use-case for page fault handling (for this milestone) is `morecore`, which is the function called from inside `malloc` and friends when they run out of currently-allocated heap memory. Your task is to implement `morecore` without resorting to having a large static array of frames (as the supplied code does).

Instead, you should now be able to just *reserve* a region of virtual addresses as the heap (using `paging_alloc`) and return one of these addresses as new heap space for `morecore`, without acquiring any physical memory for them. Following this, when the application touches an address in this new range, it will take a page fault, and only then will the page fault handler allocate physical memory on demand to back the reserved virtual addresses.

One test scenario for this is to `malloc` a very large array (e.g. 100MiB), but only touch a few bytes in the middle. This should only require a few pages of physical memory to run successfully.

Remember that there may be multiple threads accessing the heap of your program at the same time. You will need to take care that you are handling a page fault for a heap page only once and don't install a frame twice.

Once you feel confident that your page fault handling implementation is solid, you can change the libaos initialization code to run the main function for regular domains on a dynamically allocated thread. To enable this behavior, do the following two things:

- `#define` the symbol `SELF_PAGING_WORKS` at the top of the file `lib-aos/threads.c`.
- `undefine #define USE_STATIC_HEAP` at the top of `morecore.c`

## 6.10 Dynamic stack allocation

You can use the same mechanism to catch accesses beyond the bottom of a thread's stack, and extend it by allocating more memory. Note that *moving* the stack is difficult, so you want to make sure that each thread's stack has plenty of empty virtual address range to grow into.

If a page fault occurs just off the end of the currently allocated stack for a thread, you can detect this (i.e. notice that it's a stack fault) and allocate another page to back the new stack frame, up to some stack limit (beyond which it's a real error).

Most OS designs have this functionality for allocating memory even for the main thread stack, which grows down from high addresses instead of being allocated on the heap.



Extra  
Challenge

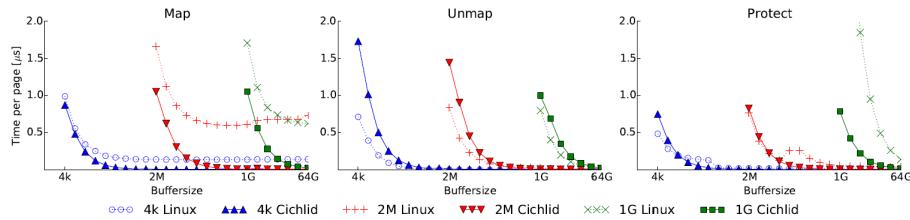


Figure 6.4: Comparing VM operations in Barreelfish (Cichlid) and Linux 4.2.0

## 6.11 Dynamic capability slot and data structure allocation

We've glossed over an additional source of complexity in the memory system: in order to acquire more physical memory, you need to store capabilities to the new memory. For this you need capability slots in your Capability Space (CSpace). By now, you may have experienced a situation where you have run out of slots in the CSpace.

Moreover, handling a page fault may require the allocation of additional data structures to hold page-table bookkeeping information.

The solution, of course, is to allocate more physical memory, retype it to a L2CNode, and add it to your L1CNode using a free slot in that node. You allocate the memory in much the same way as you would allocate frames to service a page fault.

However, it's not quite that simple. You need to make sure you have a free slot to store the capability to the memory you need to expand your space of free slots. Just calling `malloc()` to get additional memory does not necessarily work as this memory may not yet be backed by physical memory. And so on ...

The right way to do this is to bound in advance the maximum number of free capability slots you will need to service a page fault and to make sure you have enough memory to hold additional bookkeeping information. Then make sure that you *always* have this many slots free and enough memory available. If it looks like you're about to run out, allocate another L2CNode before it's too late or grow the number of slabs.



### Background

Is self-paging worth it? It certainly has a number of attractive features, not least the precise control over what the MMU does. For example, superpage mappings and NUMA allocation in Linux is very much a hint to the OS rather than a hard requirement, whereas in Barreelfish (and other such systems) you get to say exactly



Commentary

## 6.12 Background

what you want, and if you can supply the resources (such as physical memory), the MMU will do it.

Many virtual memory use-cases not possible, or painful to achieve, in Linux (such as page-coloring for caches, or mixing superpages and regular pages in a single application) are trivial in Barreelfish.

However, is it fast enough? After all, moving functionality out of the kernel generally involves paying the overhead of system calls. Remarkably, self-paging is mostly *faster*.

Figure 6.4 shows the latency of mapping, unmapping, and changing the protection on a buffer of memory (of varying size) using Linux 4.2.0 and the optimized Barreelfish implementation, known as Cichlid. Different lines show the latency when using different page sizes as well as different VM systems. It turns out that there are many different ways to achieve this on Linux (which is another story); in each case we've picked the fastest one.

Cichlid/Barreelfish is considerably faster than Linux for mapping and protecting, and a bit slower for unmapping pages.

There are several reasons for this, but the main one is the cost of translation: since Linux's interface works entirely in *virtual* addresses, the kernel spends a lot of time translating these to physical frame numbers. With Barreelfish, capabilities immediately provide the physical address the OS needs. The one time this works in Linux' favor is when *unmapping* an existing buffer, since then Barreelfish has to obtain the virtual address as well as the physical address.

For a different set of benchmarks, Figure 6.5 shows the operations proposed by Appel and Li for accelerating language runtimes, and in all these cases Barreelfish is much faster. This also shows that different strategies for maintaining the TLB (which are hidden inside the `switch` operation) don't make a lot of difference to performance.

Another concern is that self-paging makes life more complex for the programmer. This assignment hasn't achieved the same functionality as the Linux virtual address space, in particular demand paging to stable storage. You can implement this in a library (along with other useful functions like copy-on-write), and it works reasonably well.

It's also easy to apply different resource management policies to different regions of the virtual address space – something supported in the Mach-derived virtual memory subsystem of FreeBSD but mostly missing from Linux.

As a final word, there has been considerable interest recently in using hardware support for virtualization to expose more MMU functionality to user programs. The basic idea is that if a program is running where you would expect the kernel of your guest OS to run, it can directly manipulate the page tables in the same way [23, 24]. This works up to a point: for example, the dirty and access bits on x86 MMUs can be directly read and written by the program, making this really efficient.

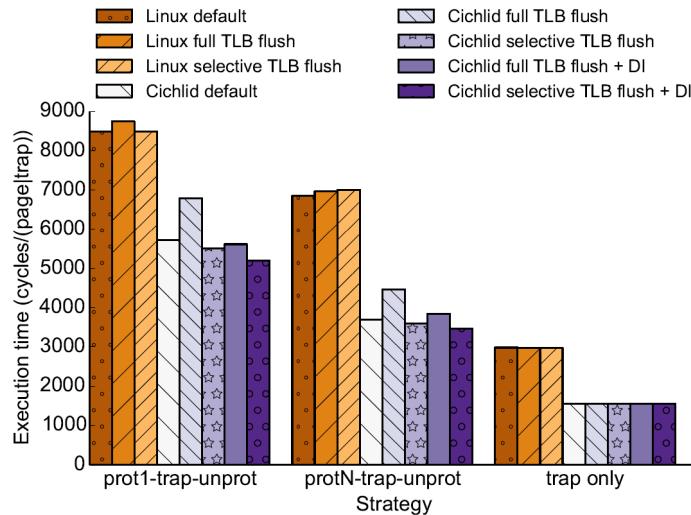


Figure 6.5: Appel and Li benchmarks [10] on Barreelfish (Cichlid) and Linux 4.2.0

However, this use of hardware support for virtualization doesn't provide the same access to memory. There is another layer of virtual address translation underneath guest page tables which the user has no control over (in contrast to self-paging, where the complete translation is controlled by the user). Moreover, the nested paging hardware involved requires more TLB entries for the same degree of coverage, which can really slow down programs with large working sets.



### Commentary

## 6.13 How we test

For this milestone, there is no specific interface to implement. Instead we will use the functions `malloc()` and `free()` to allocate heap memory and exercise your self-paging code. If you can handle arbitrary sequences of `malloc()` and `free()` without crashing or running out of memory unnecessarily, it is likely that you have a reasonable implementation.



### Commentary

## 6.14 How to do well

It's possible to do a really minimalist version of self-paging that just about works, but this is not going to score a lot of points in this assignment, or help you in the rest of the project.

To do well on this assignment, think about how you can *efficiently* implement virtual memory operations. As discussed in section 6.12 the real challenges with Barreelfish's interface lie in efficiently implementing unmapping operations. You should

be able to explain how and why your abstraction for managing virtual memory is able to accomplish the task.

There's a lot of concurrency involved in this assignment: you can have multiple threads that can call `malloc`, and/or trigger a page fault. This might happen while another page fault is happening. Think carefully about how to deal with this situation, and be prepared to talk about it.

To do really well, make sure that your implementation of `paging_map_fixed_attr()` is really solid, and can deal with (mostly) arbitrary mapping requests. This is a prerequisite for being able to handle arbitrarily many capabilities of arbitrary, page-aligned size, and is likely to make your life easier later on as the system you're building gets bigger and more sophisticated.

As always, pay attention to your coding style. Clean, well-documented and well-commented code usually gets more points.

## 6.15 Milestone 4 summary

You will be expected to demonstrate the following functionality:

- A process using your implementation to trigger and handle page faults.
- The process handling a page fault by allocating memory to back the reason, and then restarting the faulting thread.
- The `malloc()` function working with this dynamically allocating memory.
- Allocate a large (256MB) buffer on the heap and only touch a small part of it.
- Create and handle page faults from multiple threads.

Your implementation should be able to handle dynamic allocations of total size at least 64MiB. In other words, the `init` binary that we provided should not abort when this much memory has been allocated.

In addition, you are expected to be able to explain the following aspects of your work:

- How your dispatchers are taking and handling page faults
- The user-level address space storage format; what data structures do you maintain, and why?
- The fields in the `struct paging_state`, and how they are used.

Extra challenges:

- Show that you can also dynamically extend the stack in response to page faults.



Project  
Instructions



# Chapter 7

## Multicore

In this milestone, you will bring up another CPU core in your system, establish communication between the two cores, and run applications on the second core. The focus in this chapter, other than booting the second core, is on how to manage and coordinate memory and other resources between the cores, and the communication channel you will build will be fairly simple. Later on in the next chapter we'll make it a bit more sophisticated and also look more into the history of multiprocessor communication.

These days multicore chips are almost pervasive – it can be quite a challenge to buy a processor that does not have more than one core. A typical processor chip has at least two cores for running applications, plus a bunch more that you probably don't see or at least are not managed directly from Linux.

### 7.1 A concrete example: the Toradex board

For example, the NXP i.MX 8M you are using has four ARM Cortex-A35 cores which you're programming in this course, and so far you have only run code on one of them.

However, if you go back to [Figure 1.7](#), you'll see a lot more. There are two Cortex-M4 cores in the system: one is part of the System Control Unit or SCU, which is used for power management on the chip (among other things). A second (the “User CM4 Complex”) is intended for real time processing and low-power tasks depending on the application.

Beyond this, there are several other cores visible. For example, the SCU also contains a Cortex-M0 core acting as a security processor. There is a set of Digital Signal Processor (DSP) cores for audio and video codecs, and a Graphics Processing Unit (GPU) for graphics and other tasks.



*Background*

In fact, it's even more complex than that. This chip, and many of the other chips on the Aster board, have a plethora of embedded controllers on them which are, technically, programmable, but whose programming interface is either frozen when the board is manufactured, or which are programmed only by other embedded processors in the system (such as the SCU). The implications are that there is an awful lot of software and hardware in a modern computer system in which the so-called operating system (like Linux) is not involved in any way. If you think this might be a security concern for some people, you'd be right.

When using QEMU, you may configure it with many virtual CPUs (vCPUs). Your host machine will schedule them on the real CPUs of your machine. It is possible to have QEMU use more vCPUs than there are real CPU cores available. You can configure the number of vCPUs using the `-smp` argument.



#### Background

## 7.2 Early multiprocessors

In 1997, the example we've just seen would be almost unthinkable. The vast majority of microprocessor CPUs had a single core.

That said, *multiprocessors* (computers with more than one processor) have a long history in computing prior to the rise of *multicore* (more than one processor core on a die), though it was mostly in niche areas and research. Many of the fundamental ideas in multiprocessing were anticipated by the research community long before they became mainstream. Commercial uses of multiprocessors were typically limited to "big iron" (mainframes and supercomputers), and relatively few OSes designed from the outset for multiprocessor hardware.

This is important to bear in mind when reading OS research papers from the 1990s and earlier: if the paper doesn't mention that the hardware is a multiprocessor, you are probably reading about a system which is implicitly designed for (and evaluated on) *uniprocessor* hardware.

Multiprocessors were only needed when a single processor wasn't fast enough, and the performance gain justified the considerable cost in extra software and hardware complexity.

A true multiprocessor OS can be defined as an OS which runs on a tightly-coupled (usually shared-memory) multiprocessor machine, and provides system-wide OS abstractions to programs. We'll describe a couple of these below, but there were many more important ones.



#### Background

## 7.3 Multics

Any history of computing has to mention Multics sooner or later. It's a popular joke that every new idea proposed in OS research had already been done in Multics, and probably didn't work.

Multics was an ambitious time-sharing operating system for a multiprocessor mainframe, developed as a joint project between MIT, General Electric, and (until 1969) Bell Labs. Multics systems ran with real paying users from 1965 to the mid 1980s, and the last Multics system was actually decommissioned in 2000.

The aim of the Multics project was to build a computing “utility” shared between many different paying users (today this is known as “cloud computing”). The engineering goals were reliability, dynamic reconfiguration (for when things did fail), security (between users and their data as well as protecting the OS from malicious programs), and many others.

Multics was hugely influential. It is true that many key ideas in naming, binding, security, virtualization, storage, linking, etc. were first tried out in Multics. The canonical reference for the system is E. I. Organick’s book [85], but in practice the system was rather different from that described in the book. The web site [multicians.org](http://multicians.org) is a collection of documents and reminiscences from people who worked on the original system, and is closer to the truth.

Multics ran on a GE645 computer, which looked rather different to a multicore PC or phone today. It was a symmetric multiprocessor: all processors had shared access to the same main memory, and was a Uniform Memory Access (UMA) machine: the cost of accessing any part of main memory was the same regardless of the accessing processor or physical address.

Processors communicated with each other using “mailboxes” implemented in the memory modules, and by corresponding interrupts. Communication was said to be *asynchronous*: the sender wrote message payloads into mailboxes, and then notified receivers using interrupts. The hardware had a reliable interconnect, with no caches.

Among many innovative ideas, Multics implemented a “single level store”. Disk (and drum) storage was addressed using the same physical addresses used for RAM. The virtual memory hardware was based on *segments* (and each file was represented by a different segment), with demand paging within each segment. All code and data was dynamically linked and shared between processes via an ingenious system of special “linkage segments”.

The Multics hardware could be reconfigured online into multiple separate systems, and then recombined later: Multics systems were regularly partitioned into operational and development/testing systems and then merged without shutting down the system.

Multics was slow, but its influence is everywhere in operating systems and hardware design. The protection and addressing model of the x86 architecture is heavily based on Multics (with its use of segments, paging, task descriptors, and ring-based protection), as was the PA-RISC architecture [63], which was designed in part by former multicians.

Likewise, many ideas in how to build an OS for a shared-memory multiprocessor were either lifted from Multics, or developed as a reaction against it.

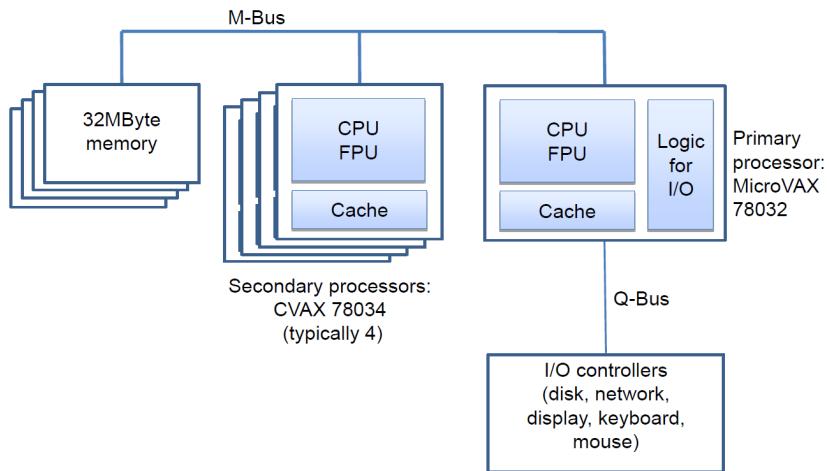


Figure 7.1: The Firefly multiprocessor architecture

Famously, Multics was so complicated that when Ken Thompson and Dennis Richie wrote a tiny simple OS to support them playing Spacewar in their lunch break at Bell Labs, they named it **UNIX** as a pun on Multics.



#### Background

The Firefly was a workstation (a personal computer that sat beside the desk) built around 1988 by Chuck Thacker and others at Digital Equipment Corporation (DEC)'s Systems Research Center in Palo Alto, California. We have already seen work done in the context of the Firefly and its OS, TAOS, in previous chapters: Scheduler Activations and Lightweight Remote Procedure Call.

The Firefly [100] was an extremely powerful personal computer for its day. It was one of the first multiprocessor workstations, and consisted of between 3 and 9 VAX processors [71]. At the time, DEC sold departmental minicomputers using 1 or 2 of these processors, and high-performance graphical workstations with a single VAX chip.

Figure 7.1 shows the machine's architecture. The Firefly was a UMA machine with early hardware support for cache coherence (it used a MESI-like protocol). Caches were small, and the use of a single memory bus meant that contention for memory and the interconnect were important issues. Even so, analysis using trace-driven simulation and a simple queuing model found that adding processors improved performance up to about 9 processors.

TAOS, the OS written for the Firefly, was highly thread-based, and pioneered many

techniques now used in mainstream multicore OSes. We'll see more of the research done using the Firefly in the next chapter.

Note that, strictly speaking, the Firefly was a *heterogeneous* multiprocessor: the main processor was a different chip to the secondary cores. The instruction set architecture was the same, but in some cases it mattered which core a piece of OS code was running on.

## 7.5 Multicore



Background

The systems we've discussed so far, and many others around at the time, were not mainstream systems. They were mostly there for research purposes. DEC built a lot of Fireflies, and they were used extensively inside DEC research labs and universities as people's everyday desktop machines (I used one as a student), but they were never marketed as a product).

Moreover, the processors on these machines were all single core microprocessors with one socket per processor (in the more recent hardware, like the Firefly) or each processor was itself built from discrete components (as in Multics).

This all changed with the arrival of multicore chips in the early 2000s. As we have seen, this was prompted by the halt in any performance improvements due to increasing clock frequency or Instruction-Level Parallelism (ILP), but the continuation of Moore's law. This meant that the easiest way for processor vendors to continue offering improved performance was to put multiple cores on each chip.

This was a big deal at the time, because most mainstream software up to that point was written to be single-threaded: it was easier (meaning you didn't have to pay programmers as much) and it ran just as fast as multithreaded software since you only had one processor to play with. Moreover, the performance of this software increased over time *without the programmers having to do anything*, since the typical processor executing the code just got faster over time. Before multicore chips appeared, the joke was that the easiest way to increase the performance of a program by 20% was to go on vacation for 12 months.

Multicore changed this because multiprocessors were now the norm instead of being niche computers, a lot of software had to be rewritten to be multithreaded, and any serious applications programmer (as opposed to systems programmers, who were already familiar with this problem) now had to be an expert in parallel and concurrent programming. This, incidentally, is why ETH Zurich and some other top Universities started to teach parallel programming to first-year undergraduates rather than Masters students.

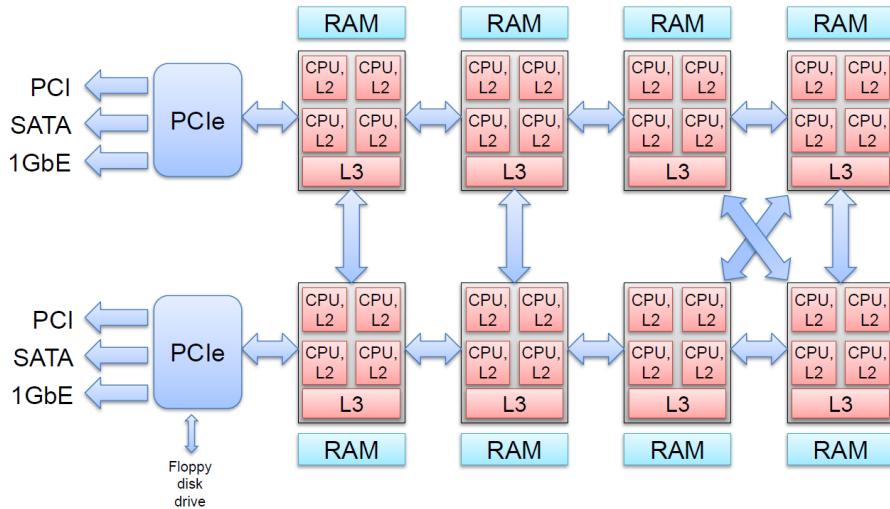


Figure 7.2: A large multicore PC, circa 2007



### Background

## 7.6 Scaling an OS to multiple cores

What did this mean for mainstream OSes like Unix, Linux, VMS, and Windows? These systems had been handling multiprocessors for some time before multicore processors arrived, but the challenges of evolving an OS which was originally designed for a single processor system to one with many cores are considerable, and still being worked on today.

The first problem is one of scalability. Once you've made a uniprocessor OS kernel work without crashing on a multiprocessor (typically by putting a big lock around the entire kernel), you find that on, say, a 16-core machine it runs nowhere near 16 times as fast as on a single processor and, in some cases, slower. The more cores a machine has, the bigger this problem of scaling performance is. Figure 7.2 shows a PC server that we purchased in 2008, which even then had 32 processors spread over 8 sockets. Note that it also had a floppy disk drive.

The first scalability improvement to be implemented in systems like Linux was to replace the One Big Kernel Lock with progressively finer-grained locking. This is complex and tricky to get right: there end up being lots of locks in the kernel, and they have to be acquired in the right order to prevent deadlocks. Nevertheless, this does improve scalability significantly. All major monolithic OS kernels today have fine-grained kernel locks, though interestingly for *microkernels*, there is some evidence that One Big Kernel Lock is better in some cases (since in a microkernel it's just not that Big) [86].



Background

## 7.7 Scalable locks

Doing fine-grained locking well soon exposes the next scaling problem: the locks themselves don't scale. Consider the following naive spinlock implementation:

```

1 void acquire(int *lock)
2 {
3     while( TestAndSet( lock ) == 1 )
4         ; // Spin
5 }
6 void release(int *lock)
7 {
8     *lock = 0;
9 }
```

This, of course, doesn't work in its current form, but let's forget about the problems with memory consistency models for a moment (we'll come to those in the next chapter) and assume it works. The problem is that the `TestAndSet()` call typically locks the memory bus (or the memory controller in a more modern machine) and slows everything down. To avoid hammering the memory system with expensive Read-Modify-Write cycles, people used "test-and-test-and-set", e.g.:

```

1 void acquire(int *lock)
2 {
3     do {
4         while( *lock == 1 )
5             ; // Spin
6     } while TestAndSet( lock ) == 1 ;
7 }
8 void release(int *lock)
9 {
10    *lock = 0;
11 }
```

This, also, doesn't work in this simple form except on a machine with a ridiculously strict memory model (and certainly won't work on ARM, as we will see), but even if it did, it still has a scaling problem when the lock is contended: when it is released, every processor which is waiting to acquire the lock unleashes a Read-Modify-Write cycle at the same time, jamming up the memory system. A large number of cache coherency messages are generated, which can end up serializing execution *inside* the memory system, killing scalability (recall Amdahl's law). This is ironic since we know (and hope!) that only one processor will obtain the lock.

A classic solution to this problem is the Mellor-Crummey Scott (MCS) lock [80], and numerous variants that have appeared since. The key idea of the MCS lock is that each thread that is spinning on the lock should spin on a *different* memory location – in fact, a different cache line – and the holding thread should release the lock by modifying only one of these, letting only one thread at a time see that the lock has been released and that it can now acquire the lock.

The code for `acquire()` looks like this:

```

1  struct qnode {
2      struct qnode *next;
3      bool locked;
4  };
5
6  typedef struct qnode *lock;
7
8  void acquire( lock *l, struct qnode *qn )
9  {
10     struct qnode *prev;
11     qn->next = NULL;
12     // Exchange with the previous tail of the list
13     prev = FetchAndStore(l, qn);
14     if (prev != NULL) {
15         // Queue was non-empty; we need to join the queue and spin.
16         qn->locked = true;
17         prev->next = qn;
18         while( qn->locked )
19             ; // Spin
20     }
21 }
```

Note that `acquire()` takes an extra argument, which is a structure which the calling thread will spin on – it will keep examining this data structure to see if it has acquired the lock. This is allocated by the calling thread; it needs one of these for each lock it wants to acquire at a time.

The lock data structure itself is a linked list of such polling structures; the `lock` argument `l` is a pointer to the *last* element of the list. The lock is not held if this tail pointer of this list is `NULL`. The thread calling `acquire` atomically adds itself to the tail of this list using an atomic exchange operation, and then spins until it becomes the head of list, at which point it has the lock.

The `release()` side looks like this:

```

1  void release( lock *l, struct qnode *qn )
2  {
3      if ( qn->next == NULL ) {
4          // Seems to be nobody waiting
5          if CompareAndSwap( l, qn, NULL ) {
6              // There really was nobody waiting; nothing to do
7              return;
8          }
9          // Wait until someone is definitely enqueued.
10         while( qn->next == NULL )
11             ; // Spin
12         // Release the lock
13         qn->next->locked = false;
```

14 }

MCS locks scale really well, since they only spin on thread-local variables. In addition, they provide FIFO ordering, require a small constant amount of space per lock, and also work with or without cache coherence.

MCS locks work by minimizing the sharing of data (the lock words) between cores – strictly speaking, between *caches* – and restricting such sharing as much as possible to only pairs of nodes. In this way, they minimize the inter-cache traffic between nodes when the lock status changes. Variations of such “scalable locks” target different workloads and hardware configurations [36].

This, however, points to a bigger problem: if the memory words or cache lines which form the lock itself can become a scalability bottleneck when too many cores try to access them, what about the data protected by the lock itself?

MCS locks also point to a way of dealing with this problem of the cost of sharing data: just don’t. Instead, either partition OS data between cores, or replicate it across cores. In the latter, the replicas have to be kept consistent, just as in a distributed system. Exactly what “consistent” needs to mean depends on the use case.

## 7.8 Tornado and K42

This recognition that simply sharing data, and in particular data that is modified, is a problem for scaling and performance in multiprocessor OSes was first addressed in Tornado [49] at the University of the Toronto and its successor at IBM, K42 [67].

Tornado used *object orientation* as a way of decomposing the functionality of the OS. It may be surprising to use objects as a way to improve the performance of the system, but by encapsulating the state of OS entities like open files, virtual address spaces, etc., it reduced the sharing of state between independent tasks in the system.

Figure 7.3 (slightly modified from [9]) shows an example of two different memory-mapped files in an address space. Different code paths executed on different cores, for example operations on the different address space regions, for the most part touch disjoint objects, and therefore do not share cache lines.

Object-oriented decomposition reduces sharing of cache lines between different OS objects, but does not by itself address the problem of sharing cache lines between different cores accessing the *same* object. In this case, where state has to be shared, it must be replicated between cores, and the replicas kept sufficiently consistent. Moreover, exactly *how* and *when* the state should be replicated depends on the workload and particular use case.

This leads to a new problem: how can the *interface* of a replicated object accommodate such changes in replication. This can be hard to program.



*Background*

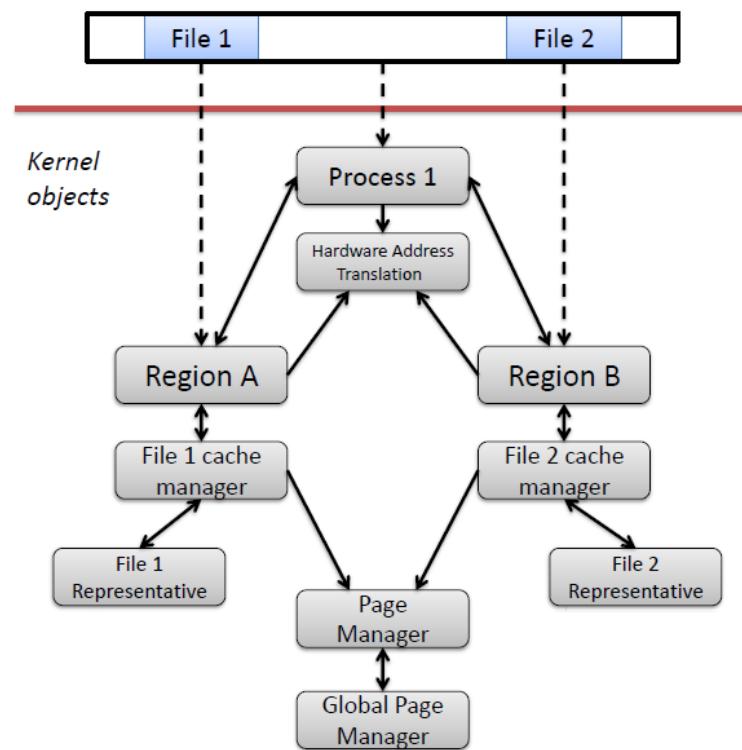


Figure 7.3: An address space with two mapped files in K42

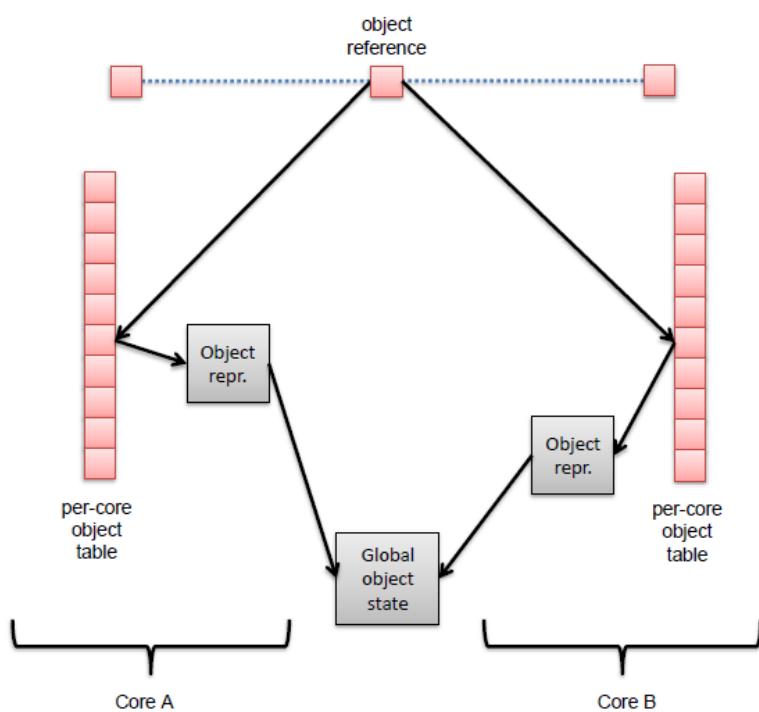


Figure 7.4: A clustered object

To address this, *clustered objects* were used, which extend the idea of separating code paths and state inside an object. References to each object in the system are indirection through a per-core object table (C++ smart pointers make this process transparent) to a core-local “representative” of the object, which has references to any state of the object shared between representatives. Figure 7.4 shows abstractly how this worked. The same reference on different cores pointed to a different representative.

Consider a very simple example of a clustered object: an integer counter. If we expect that the counter is going to be frequently read, but rarely incremented, it makes sense (on a cache-coherent machine) to centralize the state since it can be read-shared most of the time. Increment operations must then take out a spinlock on the counter value, or use an atomic increment operation. Both require some kind of global coordination, and more importantly invalidate all cached copies of the counter value, but this is a rare operation.

On the other hand, if we expect the counter to be frequently incremented but relatively rarely read (for example, if it is collecting statistics, or keeping track of some logical timestamp), we can do much better. If every per-core representative of the counter has its own private count value, this can be incremented independently of any other cache. The cost of this, of course, is that to read the value we need to take out a global lock and total up all the private values.

Both these two implementation strategies have the same interface, and in K42 they would look just like a single-threaded counter class object. However, they would perform (and more critically, *scale* with core count) very differently.

Stepping back, we’ve seen the trend of increasingly fine-grained locking in search of multicore scalability, followed by the use of replication (initially of lock state, later of any contended OS state) as a transparent optimization of shared data structures.

K42, and Tornado before it, influenced Barreelfish considerably. It also had a similar influence on Linux which has increasingly incorporated per-core datastructures (such as scheduler queues and slab allocators). However, Barreelfish itself takes a different approach to scaling.



## Background

### 7.9 Scalable commutativity

K42 was designed to run on a large IBM Power Architecture multiprocessor, but provide a Linux-compatible binary interface to system calls. It succeeded in this goal, but at considerable cost.

The first problem is that implementing a fully POSIX-compliant system is a huge amount of mostly boring engineering, and Linux compatibility is even more work. Add to this the fact that the Linux interface is not specified, and changes with kernel versions, and it becomes clear that full compatibility is beyond the capabilities of a research group (even in IBM).

The second problem is in some ways more interesting, however. If we look at the example in 7.3, some objects are shared between code paths even though the operations are on different file objects. This is imposed by the POSIX interface itself: operations on memory mapped files go through the file descriptor table or the address map, and it's unclear how to avoid this. The interface design is constraining the implementation and forcing state to be shared, where a different interface design would not require it.

Pinning down exactly what's going wrong here is subtle. There has been considerable theoretical work on multiprocessor programming and sharing [57], but what this means for the OS interface was only identified recently in the form of the "Scalable Commutativity Rule" [34].

The rule states, informally, that where several API operations *commute*, it is possible to build an implementation which is free of sharing conflicts, and therefore can scale with the number of cores. Operations commute if their execution order cannot be distinguished from the interface.

One example of non-commutative operations in POSIX (and Linux) is the allocation of file descriptors. Calls like `open()` and `create()` are specified to allocate the lowest unused file descriptor number, meaning two successive calls to `open()` on different files will behave differently if executed in a different order. This prevents a scalable implementation; if the system could assign arbitrary numbers, one could partition the file descriptor space between cores and prevent conflicts.

Another example from Unix is that local Unix domain sockets are required to order all messages send on them, even from different threads. This means that successive `send()` and `recv()` calls on a socket do not commute on a socket. Relaxing this requirement would allow scalable Unix domain datagram socket communication.

## 7.10 Barrelfish

One of the key original ideas in Barrelfish was to address the sharing problem head-on by avoiding sharing as much as possible. Broadly speaking, the difference between Barrelfish and K42 can be summarized as follows: K42 introduces replication and distribution of data as an optimization (under the covers) of a shared-memory model, whereas Barrelfish starts with the assumption that *no* operating system state is shared at all – it is all distributed or replicated, and the interface reflects this.

This is a somewhat radical position, but is arguably the logical extension of the trend we identified earlier. Barrelfish has almost no shared datastructures between cores at all - each core runs a completely separate CPU driver. CPU drivers themselves do not communicate with their peers either – each one runs in isolation. The capability system keeps track of which datastructures are associated with which cores.



*Background*

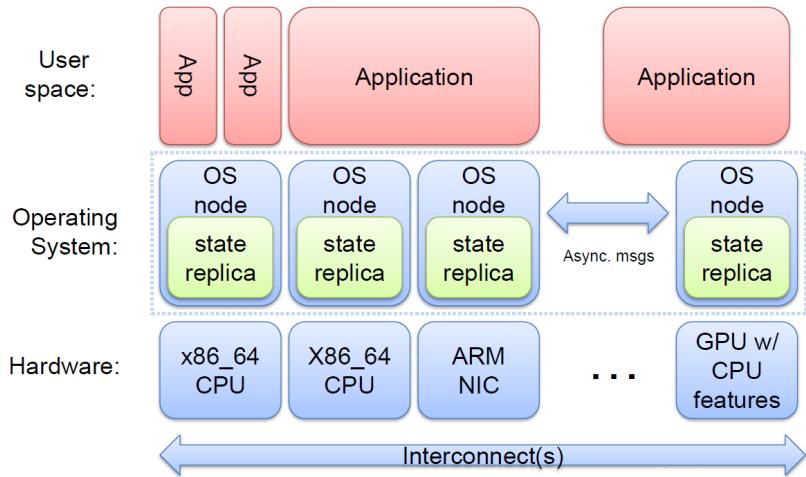


Figure 7.5: High-level Barrelyfish multikernel architecture

The structure is shown in Figure 7.5. Above each CPU driver are dispatchers and these are what communicate between cores, by whatever means necessary. In many cases, this is where shared memory *is* used, but not in all cases (such as where it doesn't exist!).

A special dispatcher on each core called the Monitor serves several purposes: it carries out long-running operations (such as capability deletion) which might take too long if run inside the CPU driver with interrupts disabled 7.6. It also communicates with monitors on other cores to exchange capabilities securely and route inter-core messages on behalf of other dispatchers.

Finally, moving functions out of kernel mode and into user space does make implementation and debugging easier – it's arguable that Barrelyfish would be faster (though not more scalable) if the CPU driver was made preemptible, and the monitor code pushed into kernel space, but it would be much more complex.

Whereas in a shared-memory based monolithic kernel like Linux (and even K42), correct operation requires making sure that updates to data structures take place under appropriate locks that preserve system invariants, in Barrelyfish this is replaced by distributed invariants that must be maintained by distributed algorithms. In the absence of failures, this might be done with two-phase commit or serializing operations on a particular piece of state through a particular core. As the hardware gets bigger and partial failures become a concern, you need state machine replication via Paxos or something similar. We're not there yet, but it's beginning to happen.

This architecture is called the *Multikernel* [21] (a term coined by Andrew Baumann).

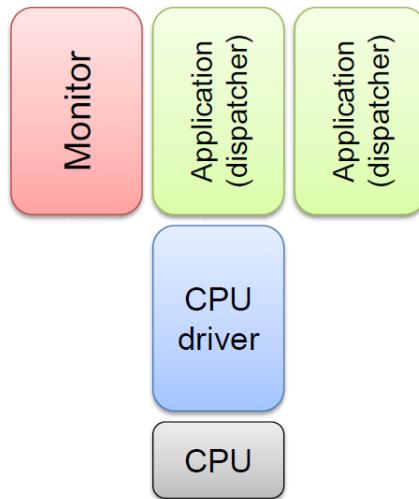


Figure 7.6: Per-core Barreelfish architecture

It is not unique to Barreelfish – other research systems like Tessellation [78] its fork, Akaros [93], and fos [107] adopted similar architectures about the same time as Barreelfish, but the latter probably pushed the idea much further. Older systems like the Auspex [30], Hydra [111], Psyche [62], and Hive [32] also share some of the same characteristics.

Almost all mainstream operating systems these days use shared-memory data-structures to the extent that it's easy to forget that alternative designs (for example, the Cambridge Tripos OS that found its way into the Commodore Amiga) were based around message passing entirely. Lauer and Needham [70] long ago pointed out that these models were duals of each other, and equivalent modulo performance issues which depended on the underlying hardware. One can argue that hardware has favored shared-memory designs since the 1970s, but the emergence of multikernels is evidence that this is changing.

All this is not the same as saying that operations must commute. Barreelfish was designed some time before the Scalable Commutativity Rule was articulated, and there are certainly places where its interface could scale better.

Of course, there is nothing to stop Barreelfish from re-introducing shared memory as an optimization (under the covers) of distributed state access. One example of this is Andrei Poenaru's work on running the same CPU driver (under a big lock) on multiple cores (or, more usefully, SMT threads) [88]. You can view this as a multicore-capable CPU driver, or instead as a CPU driver which can allocate variable numbers of SMT threads (Kathryn McKinley and others refer to these perhaps

more usefully as “lanes” through the processor [112]) to running dispatchers. The CPU driver itself remains single-threaded and non-preemptible.

This is why bringing up a new core (as you do in this Chapter) in Barreelfish is really a case of booting a completely new kernel on that core. Indeed, there is no reason it needs to be the *same* kernel implementation – we can specialize kernels for different tasks [113].

Going to a fully distributed OS design, even on a single machine, has a lot of benefits besides making it easier to scale [22]. Indeed, as we reach the limits of Dennard scaling (and hence the number of on-chip transistors that can actually be powered up at once), it’s not clear that scalability in a shared-memory system is the most important challenge. Other, perhaps more important benefits of structuring the OS as a distributed system include:

- It can easily handle cores with different instruction sets (the problem becomes one of building the code for different architectures at the same time, which is the reason Hake was devised).
- Memory that isn’t shared, or is shared but isn’t cache coherent, or is shared but at different addresses (all situations that occur on machines today) pose no great difficulties. As long as messages can be passed between the cores, the OS design works fine.
- Handling cores that come and go in the system is still difficult, but it becomes a distributed systems problem and many techniques from the distributed computing literature can be applied.
- Performance tradeoffs due to distributed memory, Non-Uniform Memory Access (NUMA) effects, asymmetric cores, heterogeneous memory types, inter-connect capacity and latency, etc. can all be cast as modifications of classical network routing problems.

These points may be decisive: hardware is evolving faster than existing system software can adapt – a situation unthinkable 20 years ago. Having an OS code base which is “agile” in the face of new platforms and architectures is becoming important.



*Technical  
Details*

## 7.11 Booting a secondary core in Barreelfish

Every machine is a little different but the basic idea is the same: Provide a start address, send a wake up event. But how do you bring a whole kernel up, bootstrapping allocators and so on?

Starting and stopping cores for power management happens all the time, but applications are long-lived. In Barreelfish, we actually allow programs at user level to spawn new kernels (CPU drivers) as required. Very little of this functionality needs to be implemented in kernel mode.

Here are the steps (in summary) to getting the additional cores up on the Toradex board:

1. **Allocate memory:** Recall that in Barrelyfish the (user-mode) caller supplies the memory needed for kernel-mode code to store its state. In the case of booting a core, this means RAM for the kernel .bss segment, RAM for init on the core, RAM for URPC channels (covered in the next chapter), and RAM to retype into a Kernel Control Block (KCB).
2. **Create KCB & Coredata datastructures:** The KCB object holds references to all of a running kernel's state, while Coredata provides the boot parameters to the new CPU driver. You can retype a RAM cap to initialize an empty KCB, for the coredata, we simply use a frame.
3. **Load the Boot driver and CPU driver:** Recall that the CPU driver is the term for a Barrelyfish kernel. When we boot a new core with a kernel we need to create a new copy of the kernel in RAM because each instance of the CPU driver has its own running state (stack, globals, etc.).

Barrelyfish CPU drivers assume when they start that a default kernel page table is set up and the MMU is enabled. The job of ensuring that this is the case falls to the *boot driver*, which is effectively a last-stage boot loader for the CPU driver which prepares this environment and jumps into the CPU driver. The boot driver can be re-used, but since it's small, it is also acceptable to reload it. We provide some support code for ELF relocation.

4. **Clean the cache:** The boot driver starts with the MMU uninitialized, and runs using uncached accesses. You need to make sure everything it needs is in RAM, and not sitting in dirty lines in the cache of the core you're currently running on.
5. **Call spawn:** The final step is to invoke an operation on the kernel capability which uses some platform-specific boot protocol (defined by the hardware) to start the boot driver, and by extension the CPU driver, on the core.

Now you're on your own: You'll start executing whatever binary you passed as the 'monitor' in Coredata. You need to figure out how to set up a channel to your initial core, and forward RPC calls and replies.

## 7.12 Getting prepared

It will be helpful to have working user-level threads in order to make some of the implementation for this milestone easier (e.g. polling on the cross-core channel). If your thread implementation is a bit unreliable, consider making it a bit more robust now.

The work for this milestone consists of three principal tasks:



Project  
Instructions

1. *Boot the second core*

Bringing up a core on Barrelyfish means booting another copy of the CPU driver. To do this, you will use *Coreboot*, as described in Section 7.13.

2. *Communicate between cores*

The LMP channels that you have used so far only connect endpoints on a single CPU driver. To communicate between domains on different cores, you will need to implement some form of user-level messaging, using shared memory – much like User-level Message Passing (UMP), as described in lectures. See Section 7.16.

Note that this is a simplified form of the task for the *next* milestone, in chapter 8: you might want to read through that chapter before starting this part of the milestone, and consider planning the two milestones together.

3. *Manage memory across cores*

Each core's CPU driver is completely independent. You must explicitly manage the distribution of resources (RAM) between your two CPU drivers. To do so, you can use the *kernel capability* to forge new capabilities, as described in Section 7.17.



*Project  
Instructions*

## 7.13 Bringing up a second core

Let's look in more detail about at bringing up another core on the Toradex board and running a CPU driver and application processes on it.

The firmware of the i.MX8 SoC implements the ARM Power State Coordination interface [14]. It abstracts away the low-level details of how cores are managed, and delegates this task to the System Control Unit (*SCU*). The SCU ensures that on boot only one core, the bootstrap core (*BSP*), starts executing. To startup further cores we have to call the SCU to power them on.

To start a core, the PSCI function we need is `CPU_ON`. It takes three arguments: The core id to be started (the `mpid`), a physical start address and a context argument which will be loaded in the register `x0`. Under the hood, this translates into a secure monitor call (SMC), and since only EL1 (where the Barrelyfish CPU driver runs) and higher are allowed to call the SCU, we have to delegate this task to the CPU driver. You don't have to implement this kernel functionality, it is already implemented in `kernel/arch/armv8/plat_imx8x.c`.

As the name suggests, the SMC instruction calls the secure monitor, in most cases this is the Arm Trusted Firmware (TF) running at EL3. Similar, to a syscall, the privilege level is switched to EL3 and execution jumps to the registered exception vector that then eventually calls the requested PSCI function. The PSCI handler then prepares the core to be booted including configuring the power and clock domains by invoking the SCU.

A user-space application can call this function using the `IPICmd_Send_Start` ca-

pability invocation on a Inter-Processor Interrupt (IPI) capability. This invocation takes two more arguments than the PSCI call: A CPU type (defined in `include/barrelfish_kpi/cpu.h`) and the `psci_use_hvc` flag. You should fix the CPU type to `CPU_ARM8`, the `psci_use_hvc` flag tells the CPU driver to use the hypervisor call (HVC) instruction to communicate with the SCU, but this is not supported on the i.MX8 and the flag is ignored.

```

1 static inline errval_t
2 invoke_monitor_spawn_core(hwid_t core_id, enum cpu_type cpu_type,
3                           genpaddr_t entry, genpaddr_t context,
4                           uint64_t psci_use_hvc)
5 {
6     return cap_invoke6(cap_ipi, IPICmd_Send_Start, core_id, cpu_type,
7                        entry, context, psci_use_hvc).error;
8 }
```

The IPI capability is only provided to privileged user-space processes, such as the *monitor* in normal Barreelfish, or your *init* process.

Now, using Coreboot isn't free – in keeping with its microkernel heritage, Barreelfish requires you to do most of the heavy lifting in user space, while the CPU driver just executes the truly sensitive operation (invoking the SCU). You need to prepare the new CPU driver to boot, and provide it with a certain amount of memory, and information. Specifically, you need to do the following:

1. *Allocate memory.* The new CPU driver needs the following:
  - Memory to load the boot driver
  - Memory to load the cpu driver
  - A `core_data` structure (`struct armv8_core_data`). One page, page aligned.
  - Kernel stack. 16 pages, page aligned.
  - Space to load the *init* process. This should be at least space for *init* plus `ARMV8_CORE_DATA_PAGES × BASE_PAGE_SIZE` bytes.
  - A *URPC* frame, to hold the cross-core communication channels that you will implement in Section 7.16.
  - A new KCB, which is the root structure holding the state for a kernel instance (CPU driver).

Except for the KCB, you can allocate these as separate frames, or in one contiguous block. To obtain a new KCB, you can simply retype a RAM cap, of at least `OBJSIZE_KCB` bytes into a KCB. Then you initialise your `armv8_core_data` structure, check out the comments in `include/barrelfish_kpi/arm_core_data.h` to understand the meaning of the struct members. Some fields, like the `cpu_driver_globals_pointer` are overwritten by the start invocation in the initiating cpu driver. Hence you don't have to set them in userspace.

2. *Load and relocate the Boot/CPU driver.* Finding and loading the ELF file for the CPU driver is exactly the same as for the user-level processes you've already spawned. We have provided support functions, `load_elf_binary` and `relocate_elf` in `lib/aos/coreboot.c`, which will handle the loading and relocation for you. Remember that the 'virtual' addresses referenced there are *kernel-virtual* i.e. relocated inside the kernel window.

Since you are inspecting the ELF binaries, this is a good time to also determine the entry points of the boot driver and the CPU driver. The correct boot driver entry point for use with PSCI is called `boot_entry_psci`. The physical address of this symbol is what you have to provide later as entry in the spawn invocation. The booted core will start executing at this address.

The boot driver will setup the kernel virtual address space and jump, in the virtual address space, to the CPU driver entry point. Hence you need to determine the virtual, relocated address of `arch_init`. You have to relocate the CPU driver to `ARMv8_KERNEL_OFFSET`, and hence you can use that offset to manually calculate the new address of `arch_init`. This information is an argument to the boot driver and thus has to be stored in the `armv8_core_data` struct.

3. *Fill in the `armv8_core_data` structure.* You need to fill in the appropriate fields, so that the newly-booted CPU driver knows where to find the initial task (`monitor_binary`), the memory for the initial task (`memory`), where the URPC frame is located (`urpc_frame`), the kernel command line, virtual cpu driver entry point `cpu_driver_entry` stack for the cpu driver (`cpu_driver_stack`), the correct boot magic (remember the iMX8 is using PSCI). Remember that the stack grows downwards, and you want to load the highest valid address into the stack field.
4. *Clean the cache.* As discussed in the lecture, you need to ensure that the data you've just written is visible to an uncached observer (the core will boot with the MMU off). You'll need at least a barrier, an invalidate and a clean.

You can use the cache maintenance functions provided in libaos, see the header file `include/arch/aarch64/aos/cache.h`.

5. *Invoke the kernel cap.* Use `invoke_spawn_core` to boot core 1. If you've done everything right, you should see messages from the booting kernel ("kernel.01 ...").

Note that the Barrelyfish codebase distinguishes between the BSP (bootstrap) processor and APP (application) processors. This distinction and naming originates from Intel's x86 where the BIOS chooses a distinguished BSP processor at startup and the OS programmer is responsible for starting the rest of the processors (the APP processors). Although it works a bit differently on ARM, the names still fit well enough. You should implement the stub function `coreboot` in `lib/aos/coreboot.c`

```
interval_t coreboot(coreid_t mpid,
```

```

2   const char *boot_driver,
3   const char *cpu_driver,
4   const char *init,
5   struct frame_identity urpc_frame_id)

```

## 7.14 Booting All Cores

You are required to implement support for booting a second core. However, there are a total of four cores on the board. As an extra challenge demonstrate that you can boot all main application cores of the Toradex board.

Note, just booting the core is the easy part. This will also make bookkeeping more difficult, as there is not just the “remote” core, but there are now multiple remote cores, i.e., requests need to be sent, and replies returned to the correct core.



*Extra  
Challenge*

## 7.15 Suspend & Resume

An operating system may turn off cores to save power. In this extra challenge, we want to temporarily stop the execution of the core. While this could, in principle, be achieved using the wait-for-event `wef` and send event `sev` instructions, we will again make use of the PSCI framework. Demonstrate that you can temporarily halt and resume the execution of a core. Checkout the PSCI functions `CPU_SUSPEND` and `CPU_OFF`. You may decide to implement a protocol that uses the suspend functionality, or completely turns off a core.



*Extra  
Challenge*

## 7.16 Inter-Core Communication

Having what essentially amounts to two single-core systems that happen to share a multicore platform is not that interesting. It also implies that you need to run two shells (or equivalent) on two cores to start other processes on those cores. While there might be a use-case for such ‘co-located’ single-core systems we want to have a proper multicore system where the different cores can share functionality such as the user interface.

Needless to say, implementing a shared user interface (and a proper multicore system in general) requires communication channels between applications on different cores. So in this step, you will implement a shared-memory based communication channel between the cores.

In the next milestone, you will implement a full communications facility (akin to LMP for a single core) using shared memory and lock-free queues, and we strongly suggest thinking ahead. Only basic functionality in communication is required for this milestone, but for the next milestone a more complete approach is required.



*Project  
Instructions*

### 7.16.1 Sharing a Frame Between Cores

As Barreelfish draws on the micro-kernel design philosophy, we prefer to push most functionality out of the kernel. This also means that we try and provide a form of inter-core communication which can be done directly from user-space without involving the kernel in every message exchange.

A simple form of user-space communication (not even necessarily across different cores) is to share a region of memory between the applications that want to exchange information and then read and write that region using an established protocol (for example, a ring buffer).

You should implement such a simple shared-memory based, user-mode communication channel. What follows is one way of doing it, but feel free to experiment with other designs.

To begin with, you'll need some shared memory. Recall the URPC frame that you allocated in Section 7.13. This frame is normally used in Barreelfish to establish an initial communication channel with the monitor process on a newly-booted core. We don't have a monitor, so you're free to reuse this frame for your own protocol. On core 1, the URPC frame is available in the task CNode. Also, the CPU driver will map it at an address that is defined in `barreelfish_kpi/startup_arm.h`. Make sure that you map this frame cacheable – using an uncached mapping is fine for testing, but it's cheating in the final submission.

Also, you'll hear in the lectures about ARM's weak memory model. You can't count on your stores being observed by the other core in the same order that you write them down! You will need to insert barriers as appropriate to ensure that your protocol is correct. You will need to explain your implementation to your tutor, and demonstrate that you understand why it is correct.

### 7.16.2 Establishing a communication protocol

As we now have shared memory between two applications, we should establish a protocol which enables meaningful communication between them. As our requirements from this communication channel are modest, a simple protocol should suffice.

We will run only one shell which will be responsible for spawning applications on both cores. Therefore, we only need primary-secondary communications. As the primary application can wait while a process is starting on the remote core, we can implement this communication as RPC.

Essentially, we need to support a *remote spawn* message as well as a corresponding response on this channel. In this remote spawn message, `init.0` should send the name of an application to `init.1` which should then start this application and report the status (success/failure) to `init.0`. The shell on the BSP core can send requests for *remote spawn* over LMP to `init.0` which then forwards the message

to `init.1` (using our new shared-memory communication channel) for actual execution. Similarly, responses can go back from `init.1` to the shell on the BSP core.

At this point you will have to properly handle the core argument to the process spawn RPC.

```
1 errval_t aos_rpc_process_spawn(struct aos_rpc *chan, char *name,
2                                 coreid_t core, domainid_t *newpid);
```

This will mean that you have to implement the necessary steps inside your process management system to enable spawning processes on the second core over a channel like the one described above.

Once you're able to send messages between cores, you're ready to start launching processes. You won't be able to actually do so yet, as the newly-booted copy of your `init` process won't actually have any memory available to make large allocations, and won't have access to the `bootinfo` structure, where the ELF images live.

## 7.17 Multicore Memory Management

Before you can actually start applications on an additional core, we will have to sort out another design question: *How do you manage your available memory between the cores?*

Currently, Barreelfish's design is based on a single memory management service which manages all physical memory on behalf of all the cores, and handles requests from all applications. This approach relies on the ability to *communicate across cores* and the ability to *find the process that is responsible for memory management*.

We recommend that you simplify this problem by splitting the memory between the cores, and let one application on each core provide a memory management service for applications on that core. This way, you can re-use most of your self-paging code on both cores.

There are different ways to tell a second core which part of memory it is responsible for. You might pass some information in the command-line arguments, but a cleaner solution is to use the inter-core channel that you have just implemented.

Given some way to tell the second core which address ranges it should use, what should you pass to it? Look at `usr/init/mem_alloc.c` in the supplied code. This function initializes the RAM allocator by taking regions from the `bootinfo` structure, which is passed to `init` on the BSP core by the CPU driver, describing the memory layout. There are several ways you could break memory up: split the RAM caps before adding it to your allocator, or simply allocate some memory from your pool to assign to the other core.



Project  
Instructions

In order to actually use one of these address ranges on core 1, you'll need a cap to it. But where will that cap come from? Recall that Barreelfish CPU drivers have completely partitioned state – you cannot copy a capability from a CNode managed by one CPU driver, into a CNode managed by another. So how are capabilities transferred between cores in Barreelfish? In a normal Barreelfish system, this is done through the *monitor*, a privileged user-level process, that acts as an extension of the trusted kernel, which is able to do things that the CPU driver can't, like block, or communicate over shared memory channels.

We already saw one of the monitor's special powers: booting cores. The second one we need to use now is its ability to create, or *forge* capabilities. Have a look at `include-aos/kernel_cap_invocations.h`. You'll see here both the KCB clone and spawn core invocations we've already used, alongside the `create_cap` invocation that you can use to create capabilities out of thin air. We've wrapped this (**extremely dangerous**) invocation for you, in `lib-aos/capabilities.c`, into two calls: `ram_forge` and `frame_forge`. These allow you to create RAM and Frame capabilities, given only a base address and a size. You must *only* use these operations to transfer resources between cores – as you don't have a full monitor to keep track of them, you can't safely invoke these capabilities on two cores at once, or delete them. All other capabilities should be derived from the root RAM caps that you create here, and can be used exactly as normal – only the forged caps are special.

You now have all the mechanisms you need to transfer RAM to core 1, and bootstrap allocations there. The last thing you need to do is to give your `init` process on core 1 access to the `bootinfo` struct and the caps referred therein. Note that the capabilities to this and all ELF modules are available on core 0, in a special CNode (see `include/barreelfish_kpi/init.h` and `include-aos/caddr.h`). You need to come up with a way to make this data available one core 1, using some combination of the above mechanisms.

By end of this step, you should essentially have two instances of a self-paging system running on two different cores. Congratulations!



Commentary

## 7.18 Experience with Barreelfish

Barreelfish's architecture has proved useful for a number of reasons in research, which is why we continue to use it as a research and teaching vehicle at ETH.

It showed that a multikernel architecture, even on conventional multicore PCs optimized for running monolithic kernels like Linux and Windows, achieved comparable performance to those systems. While very much a research OS, it runs benchmark programs, a web server, databases like SQLite and PostgreSQL as well as research-oriented RDBMS systems, an OpenSSH server, a TTY subsystem, a virtual machine monitor capable of booting Linux in a VM, and even Microsoft Office applications via the Drawbridge LibraryOS [89], though the latter is unfortunately only available to Microsoft employees.

It has also supported a variety of conventional and unconventional hardware platforms, such as PCs (with a mixture of 32-bit and 64-bit cores), ARMv7-A, ARMv7-M, and ARMv8-A processors, the Intel Single Chip Cloud Computer [], the Xeon Phi co-processor, and the Microsoft Beehive experimental soft core processor [99]. There are also rumors of a port to Tilera processors [4].

Barrelfish also showed that a lack of cache-coherence is not a barrier to building an efficient OS. Tim Harris once configured the popular GEM5 machine simulator [28] to emulate a multicore x86 PC with no cache coherence (and per-core caches the same size as main memory, for the absolute worst-case scenario). He then added less than 15 cache flush instructions, restricted to booting additional cores and the inter-core communication code we'll see in the next chapter. Barrelfish booted on this unusual machine first time with no problems.

## 7.19 How to do well

A lot of points in this milestone are about understanding what you're doing, and being able to explain it clearly. There are a number of design choices you'll need to make, and no single "right" answer. Instead, it's important that you can justify your decision.

For example, how are you managing memory between the two cores? Is the division "hard coded" (i.e. built into the source code of the system), and if so, how? Alternatively, can you determine the split using a command line parameter? Or is the memory usable by the second core actually allocated from the original pool? If the latter, how is a capability to this memory passed from `init.0` to `init.1` when the second core boots?

Another example of where some thought is required is the communication channel between the cores. It's based on a shared page, but how does this page come to be shared in the first place? How do both `init` dispatchers know where exactly the page is?

When sending messages, how does each side know when it is their turn to send/receive? In other words, can you describe the *protocol* that is used?



Commentary

## 7.20 Milestone 5 summary

You will be expected to demonstrate the following functionality:

- Booting the second core successfully, both the CPU driver and the `init` process.
- `init.0` and `init.1` are able to communicate.
- Applications on each core are able to handle page faults.



Project  
Instructions

- `init.0` is able to start programs on core 1, in other words you can start applications that run on the second core from the BSP core

Extra challenges:

- Booting all application cores of the platform (if it has more than 2)
- Suspend/resume functionality.

In addition, you are expected to be able to explain the following aspects of your work:

- Your user-level messaging protocol used for inter-core communication.
- How the second core knows which memory belongs to it, and how you manage memory allocation between the two cores.
- How barriers work in your implementation, and why.

# Chapter 8

## User-level message passing

In the last chapter we built a simple communication channel between dispatchers running on different cores using shared memory. In this chapter we'll push this idea further, looking at the history of shared-memory OS communication, how the data plane can be made efficient on a modern multicore machine, and how to construct the *control plane*, which sets up and tears down connections and turns out to be somewhat more complex than the data plane.

In this milestone, you will implement a full inter-core messaging and RPC system, to replace the rudimentary shared frame you used in milestone 5. Using this, you will extend your RPC system to allow processes to be spawned and managed on core 1 from a control process on core 0 (and vice-versa).

### 8.1 Traditional communication

This milestone also illustrates a feature of Barreelfish (when running on a shared-memory machine) that is very different from Unix-like system: inter-process communication between cores bypasses the kernel completely.

Figure 8.1 shows how IPC might work on a Unix multiprocessor system through pipes, FIFOs, Unix domain sockets, or another of the many IPC mechanisms available.

The sender first executes a system call to send the data, copying it into a kernel buffer and possibly blocking, causing the core it is executing on to switch to another process. The receiver also executes a system call to receive the data, with similar consequences. At some point, when both sides are in their respective system calls (and at least one, possibly both, are blocked), the cores synchronize and data is copied across or a buffer descriptor passed to the receiving core. The receiving process is then rescheduled, data copied up to user space, and the receiving system call returns.



Background

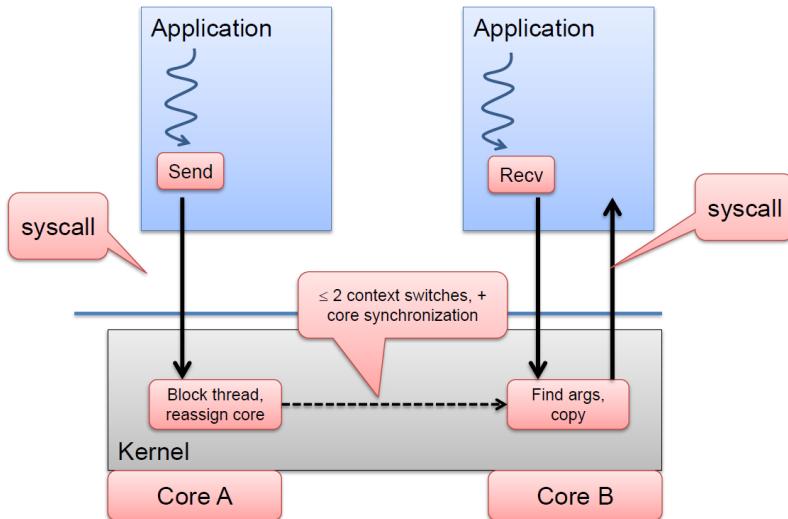


Figure 8.1: Inter-core IPC in a Unix-like kernel

This is at least two context switches, plus the overhead of two system calls and inter-core synchronization.



Background

## 8.2 Ring buffers and shared memory

In contrast, the mechanism you implemented in the previous chapter probably looked more like that shown in Figure 8.2. Two processes simply share a region of memory and read and write data into and out of it. This milestone is basically about how to refine this idea to increase throughput.

Communication between a processor and a fast I/O device like a network adaptor has long been done using *rings* of descriptors. Each valid descriptor in a ring points to a region of memory potentially containing valid data, and these descriptors are passed between the CPU and the device (using Direct Memory Access (DMA)) via a pair of producer-consumer queues implemented as fixed-size arrays of descriptors treated as rings. Figure 8.3 shows one such example, from the programming manual for the Broadcom NetXtreme Gigabit Ethernet adapter, which explains how it works on that chip – such diagrams and programming techniques are very familiar to anyone who has written drivers for network, USB, and SATA adapters or GPUs.

Ring buffers are good for high-speed I/O because they decouple sender and receiver. Both can run at their own speed with relatively little coordination. In a typical device driver, interrupts are rarely used under heavy load – they only wake up the driver when it needs to know that data has arrived, and this only happens when it has decided to stop polling the queue, itself rare when load is high.

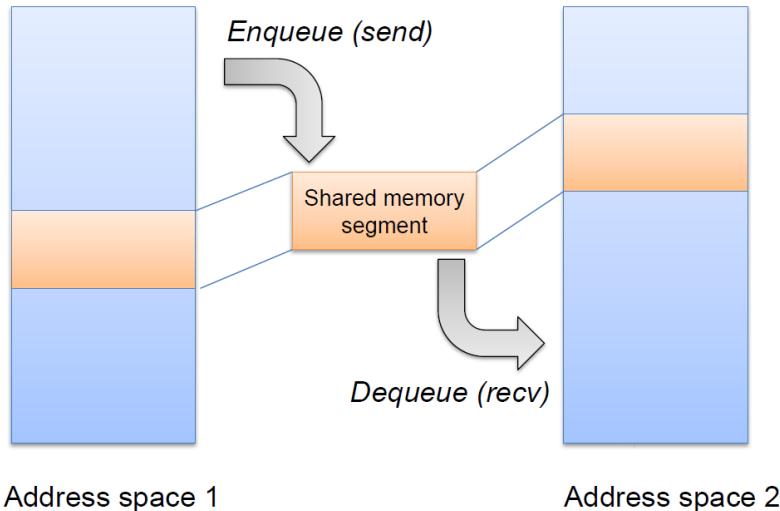


Figure 8.2: Inter-core communication using shared memory

### 8.3 User-level RPC



Background

Bershad, Anderson, et al. [27] were some of the first to apply this concept to inter-process communication in user-space on a multiprocessor, again using the Firefly multiprocessor workstation. In URPC, processes exchange data by polling shared-memory producer-consumer queues mapped pairwise between processors and protected by Test-And-Set (TAS) locks at each end.

RPC's goals were to send messages between address spaces directly without involving the kernel, and thereby eliminate unnecessary processor reallocation (and amortize the cost of processor reallocation over several message exchanges).

Arguably, URPC was to Scheduler Activations what LRPC was to kernel threads – as Figure 8.4 shows, the two are closely connected. Using the user-level scheduling provided by Scheduler Activations, messaging is asynchronous below thread abstractions but looks blocking to each thread: URPC can switch to another thread in the same address space when a send or receive is performed, rather than block waiting for another address space.

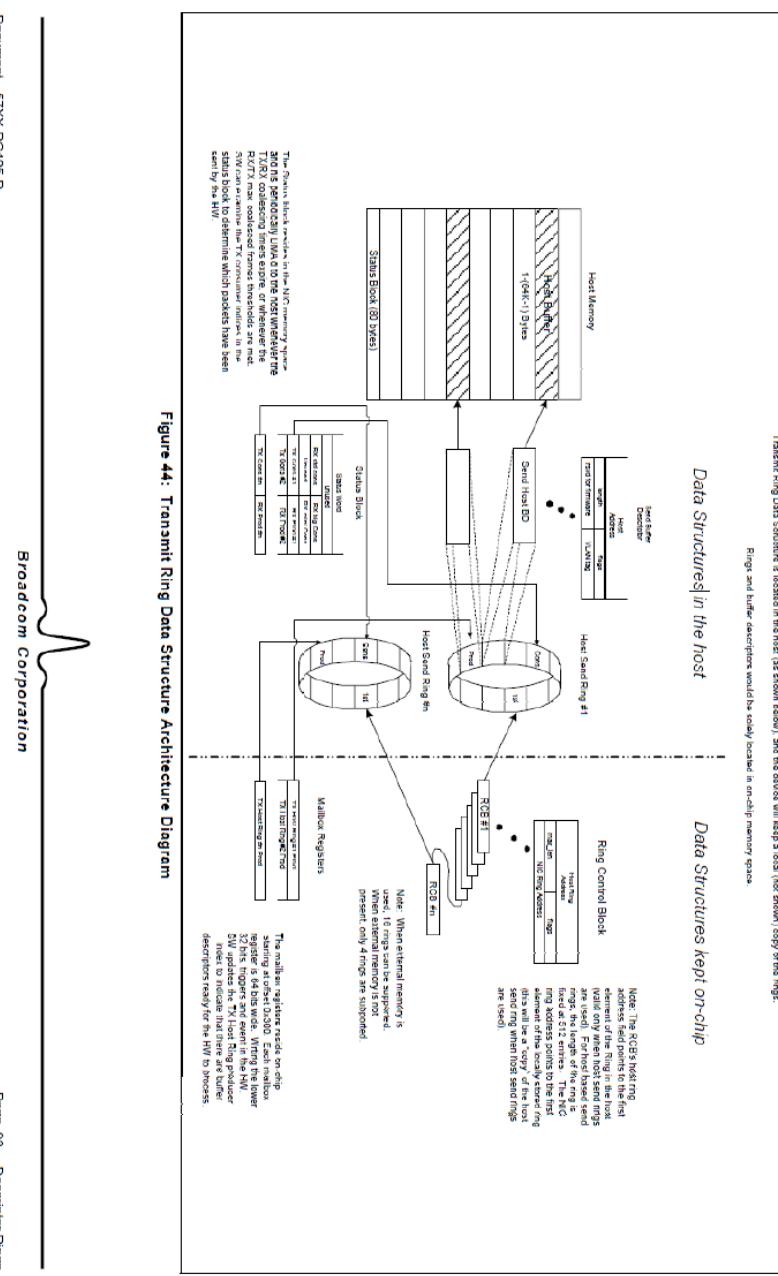
URPC works because it assumes that, in the common case where performance really matters, both sender and receiver are already running. It can then exploit the inherent parallelism in sending and receiving messages.

URPC manages to decouple three different aspects of communication:

1. **Notification**, or letting the other side know that something useful has happened, is done by the receiver on a queue polling it periodically.

**Programmer's Guide**  
01/20/08

**BCM57XX**



**Figure 44: Transmit Ring Data Structure Architecture Diagram**

**Figure 8.3: Descriptor ring documentation for the Broadcom NetXtreme Gigabit Ethernet adapter**

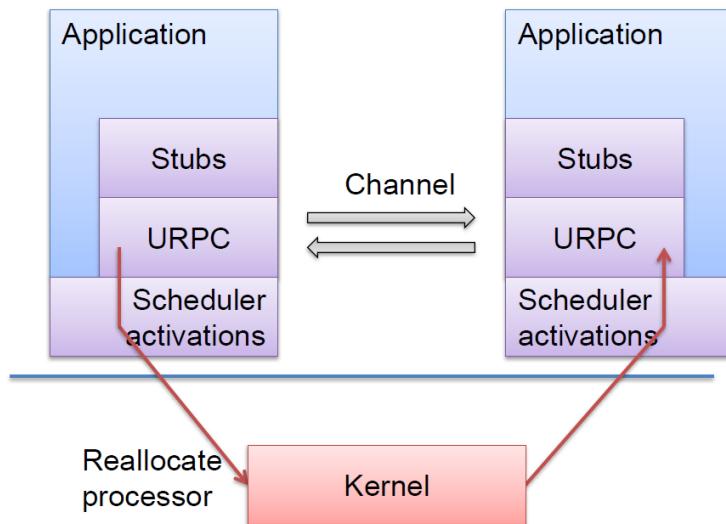


Figure 8.4: Overview of User-Level RPC

2. **Scheduling**, or deciding when the receiver and sender run, is completely separate, and is the only part of the communication handled by the kernel. In the (hopefully) common case on a multiprocessor, both sides are running at the same time.
3. Finally, **Data transfer** itself is handled by the sender writing the information into the shared memory areas.

Performance back in the day was impressive, as Table 8.3 shows.

URPC is convincingly faster than LRPC (in both latency and throughput, it turns out), and handily beats the fastest L4 RPC implementation as well. Somewhat ironically, while LRPC and L4 seek the best possible performance by heavily optimizing the kernel path, URPC attains better performance by bypassing the kernel entirely.

This is not quite the full story. Since URPC does need to poll the queues, it has somewhat high latency when the system is mostly idle. However, under load, it is much faster than kernel-based RPC systems.

On the Firefly, with very small caches and a memory system whose speed was much closer to that of the processor (in other words, the CPU/memory gap was small), URPC made a lot of sense and copying data into memory buffers was not a high overhead.

This however, begs the question: what about today?

Mechanism	Operation	Cost ( $\mu\text{s}$ )
URPC (w/ fast threads)	Cross-AS latency	93
	Inter-processor overhead	53
	Thread fork	43
LRPC	Latency	157
	Thread fork	> 1000
Hardware	Procedure call	7
	Kernel trap	20

Table 8.1: Performance of URPC, 4-processor Firefly (CVAX)

Today, hardware is very different from the Firefly. Machines are typically NUMA, not Symmetric Multiprocesing (SMP). Locking memory using Read-Modify-Write (RMW) cycles is expensive. Caches today are large, and main memory is much slower than the processors.

As a result, Barreelfish uses a different form of user-space message passing protocol which retains most of the advantages of URPC but is better suited to contemporary hardware: it doesn't use locks, it exploits the fact that spinning on a local cache is cheap, and transfers data directly between caches avoiding main memory wherever possible.

The Barreelfish inter-core shared-memory message passing mechanism (sometimes called an “interconnect driver” in Barreelfish) is called UMP. It is similar to a technique called FastForward [50], and depends critically for performance on the behavior of common cache coherence protocols.



Technical  
Details

## 8.4 A quick recap on cache coherency

One of the simplest practical cache coherence protocols for a modern multiprocessor is MESI, shown in Figure 8.5. MESI is a four-state protocol, with three valid states for a cache line:

**Shared** : The line is *clean* (i.e. consistent with main memory), and there *may* be other copies of the line in Shared state in other caches.

**Exclusive** : The line is clean, and there are *no* copies of the line in other caches.

**Modified** : The line is *dirty* (i.e. main memory has not been updated with this copy), and there are no other copies of the line in other caches.

MESI solves a problem with the simpler MSI protocol. A cache can load a line into either “shared” or “exclusive” states, and other caches see which type of read is performed (since they have to invalidate their own copy if the remote read is exclusive).

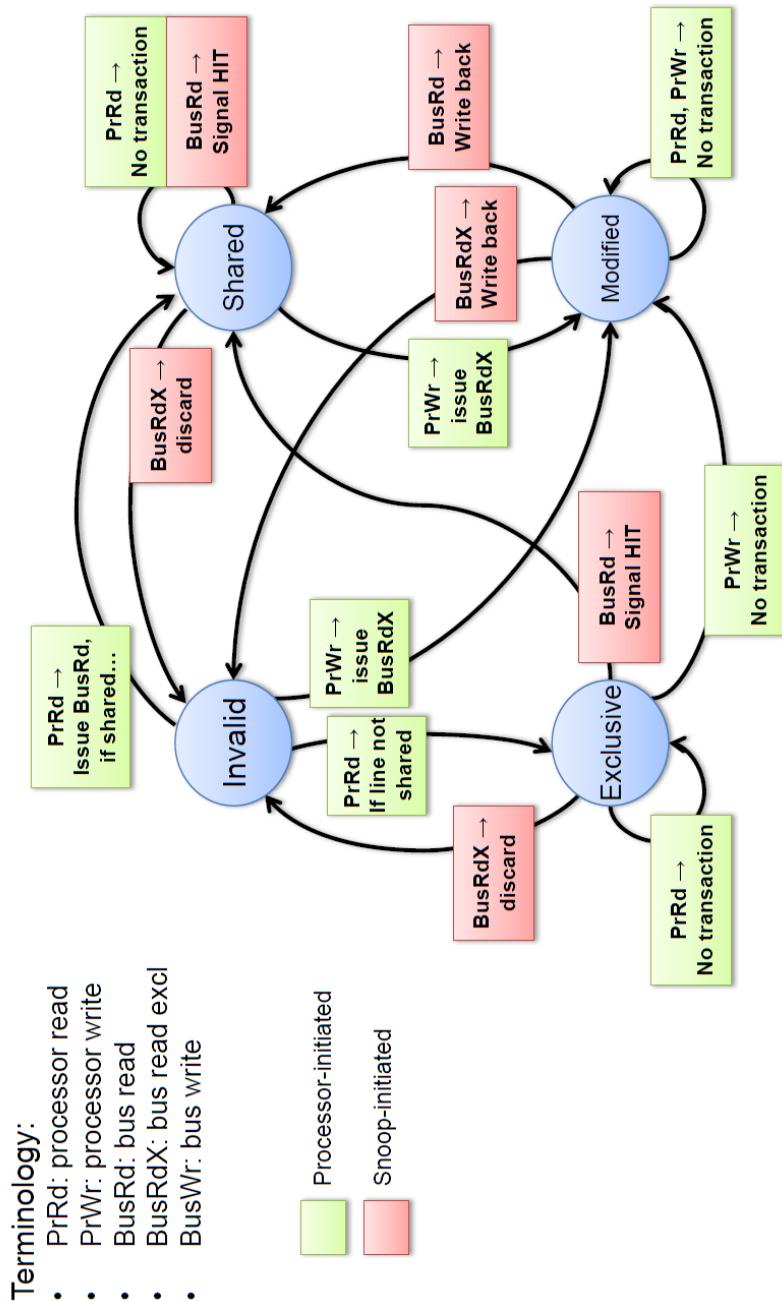


Figure 8.5: The MESI cache coherence protocol

Caches can also signal a “hit”: if they see a remote non-exclusive (shared) load and they have the line in shared or exclusive state, they can let the other cache know that it does not have the only copy.

MESI is an “invalidation-based” protocol: dirty data is always written back through memory, and is never dirty in more than one cache. No direct cache-to-cache transfers are possible. It works well if the latency of main memory is less than the latency of accessing a remote cache.

However, this is rarely the case in modern hardware, and so many processors (including AMD-based PCs and most ARM-based systems) adopt a more complex protocol which is variant of MOESI.

MOESI adds a new “Owner” state, which allows multiple dirty copies of data to exist.

**Modified** : The line is dirty, and this is the only cached copy.

**Owner** : The line is dirty, and other dirty copies might exist. They are all consistent, and this cache is the only one which can modify the data.

**Exclusive** : The line is clean, and there are no copies of the line in other caches.

**Shared** : The line *might* be dirty, and other copies might exist. All copies are consistent, and only the remote copy in Owner state can write it.

MOESI-based protocols can quickly satisfy read request for dirty cache line without writeback to memory - the Owner cache responds with the line contents. The downside is that read requests for a clean, shared line must be served by memory. MOESI works well if the latency of fetching data from a remote cache is less than the latency of reading from main memory.



Technical  
Details

## 8.5 UMP operation

We can now describe how UMP works on Barreelfish, assuming a MOESI-like cache coherence protocol. UMP is optimized for passing small messages which fit into a single cache line; in Barreelfish, compiled *stubs* can pack more data into multiple cache-line sized messages.

The basic data structure for a UMP channel is an array of cache line-sized slots which works as a circular ring buffer, as shown in Figure 8.6. The sending side of the channel maintains two pointers: one to the next slot to write a message into, and one to the last slot which the receiving side has acknowledged (and so can be reused by the sender).

Before a message is sent, the situation is as shown in Figure 8.7. The receiver keeps track of the next slot for it to read a message from, and polls this slot waiting for a message to arrive. We use the last words in the cache line to indicate that a message is valid; the receiver keeps reading this word until its value is non-zero.

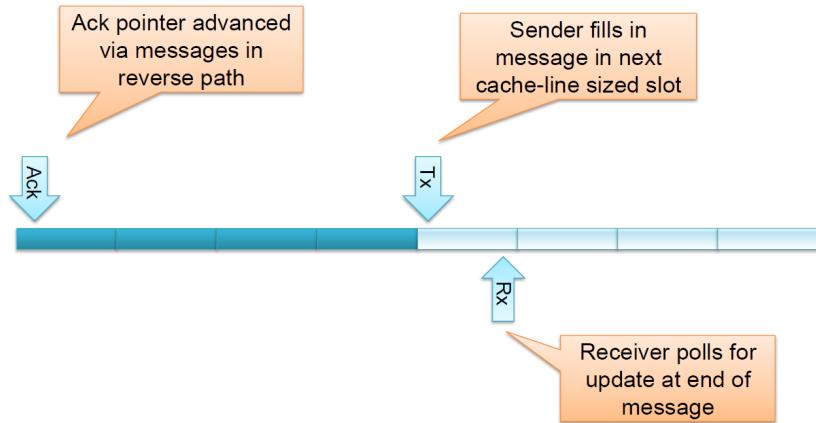


Figure 8.6: UMP message channel

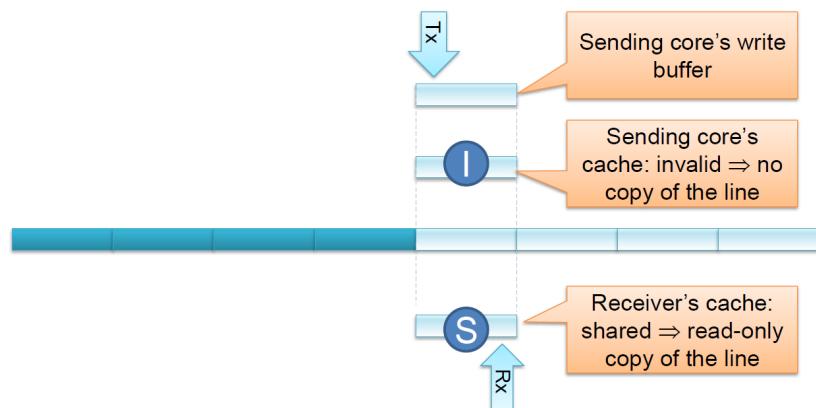


Figure 8.7: UMP state before sending a message

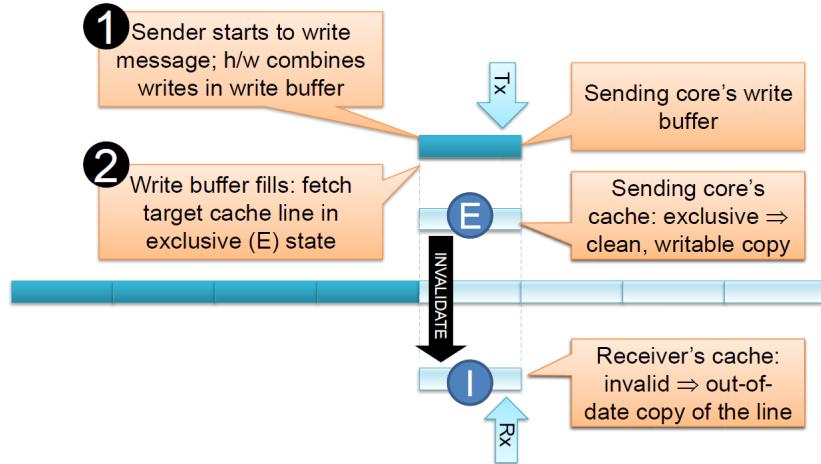


Figure 8.8: Sending a message with UMP

Since the receiver is reading the last word in the cache line, the line is typically in Shared state in the receiver's cache (indicated by 'S' in Figure 8.7). Polling this location is therefore very cheap: a load from level 1 cache only costs 2 machine cycles.

The line in the sender's cache might also be Shared, but we'll assume here that it's Invalid (the protocol works in both cases). To send a message, the sender starts writing into the cache line from the start, ending by writing the last word with non-zero metadata indicating that a message is valid.

These writes to the cache line are buffered and batched together by the sending core's write buffer, but as the write buffer fills the cache reads the line into Exclusive state (since it needs to modify its contents) (step 1 in Figure 8.8). This sends an invalidation message over the inter-core interconnect, which causes the line to transition to Invalid in the receiver's cache (step 2 in Figure 8.8).

Fairly quickly, the write buffer in the sender fills and drains into the cache line, transitioning it to Owner state (step 3 in Figure 8.9). At the same time as the sender is writing the message into the cache line, the receiver continues to poll the line by reading the last word.

This sends a probe message over the interconnect (since the line is now invalid, and must be reloaded), and the sending core's cache now responds with the new contents of the cache line, which is the message (step 4 in Figure 8.9). The last word of this cache line is non-zero, and so the receiver detects a new message and can go off and process it.

The end state is then with the sending cache holding the message in Owner state and the receiver with a Shared copy. Both these are dirty: no actual writing to main

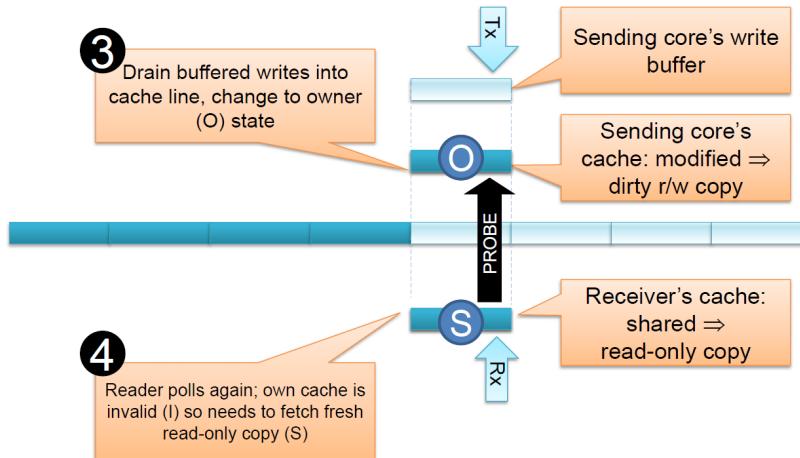


Figure 8.9: The final stage of UMP message transfer

memory has been done. We have transferred a cache line's data from one set of registers on a core to another set of registers on a different core by a direct cache-to-cache transfer without touching main memory.

This has happened with two round-trips on the interconnect (the invalidate followed by the probe), which we think is as good as it can get without additional hardware support for message passing.

Obviously main memory will, eventually, be updated when the Owner cache writes data back, but this process will not impact the latency of message transfer. Indeed, one might do several complete cycles around the ring buffer without needing to write any of the data to main memory.

For this reason, UMP is very fast if (as with URPC) the receiver is running and able to poll the channel.

## 8.6 Message passing vs. shared memory

Barrelfish's somewhat extreme position of not allowing *any* shared memory between cores was controversial, in part because the overhead imposed by message passing would be unacceptably high. After all, what could be faster than simply accessing memory?

From the discussion above, however, it should be clear that this is a dangerous oversimplification. UMP implements message passing over shared memory, but relies on the fact that shared memory itself, in a cache-coherent machine, is an *illusion* maintained by the hardware itself sending messages (invalidates, probes, etc.) between cores. Barrelfish UMP is at best half the performance of the hard-



Commentary

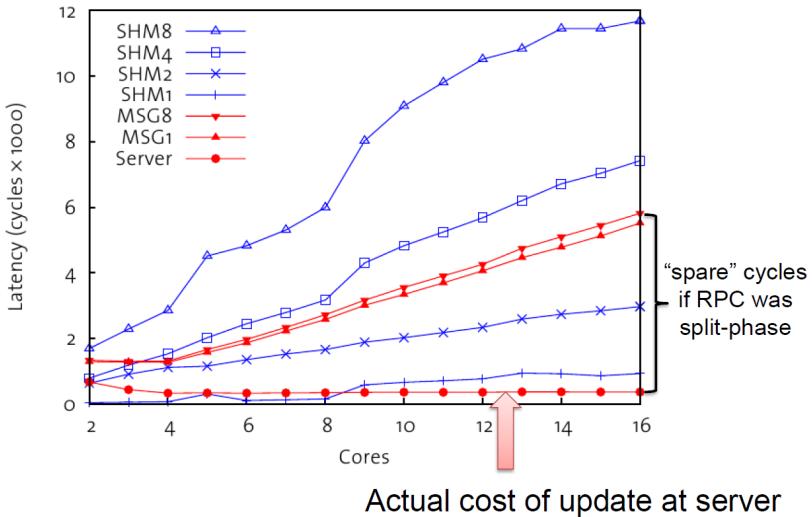


Figure 8.10: Performance of shared memory vs. messages

ware, since it needs two interconnect round-trips to transfer a cache line, but it doesn't need to touch RAM to do so (which would be much slower).

So how does this compare with shared memory? Let's try a really simple microbenchmark: we have a shared memory array of cache lines which we want to modify from threads running on different cores. We measure the latency of an operation to write to this array, which stands in for modifying a shared data structure in a monolithic kernel.

In the experiment, we vary the number of cores which want to access the data structure, and also the number of cache lines which are modified in each operation. There's no locking of any kind going on here – we're only measuring how long it takes to execute a small number of store instructions to shared memory.

Figure 8.10 shows the results for a 4-socket quad-core AMD Opteron machine back in 2009.

Line SHM1 shows the latency of a write of a word to a *single* cache line, as the number of contending cores goes up. As you can see, it's pretty fast for a small number of cores (particularly if they are all in the same socket), and increases to about 1000 cycles when all 16 cores are trying to do the same thing.

Lines SHM2, SHM4, and SHM8, however, show a different story: as the size of the update goes up (touching 2, 4, and 8 cache lines respectively), the cost gets quite large. In the extreme case, it can take a core nearly 12,000 machine cycles to perform a sequence of 8 writes to different cache lines.

What's happening is that the cache-coherence protocol is migrating modified cache

lines (as we saw with the UMP message transmission process). Note also that the processor is often stalled while a line is fetched or invalidated: there's only so much out-of-order execution even a modern processor can perform while waiting for memory. The performance of the system quickly becomes bottlenecked on the coherency protocol, limited by latency of interconnect round-trips.

Lines MSG1 and MSG8 show the latency of performing the same modifications to the data structure (1 and 8 cache lines respectively) by instead sending a message using UMP to a designated core (core 0 in this example) and asking it to do it. Everything is then serialized at this core, which is why the latency scales linearly with the number of contending cores.

The difference in cost between MSG1 and MSG8 is tiny, since all the modifications to the cache lines are now done entirely inside the cache of core 0. The overhead of doing all this over messages is about 160 cycles, which is a lot compared with memory access with only one or two cores, but look at where the lines cross SHM4: if you're modifying 4 cache lines, and the data is shared between more than 3 cores, you're faster using messages.

There's a further, hidden benefit of message passing: the actual cost of the update at the server (the "Server" line) is pretty small, since everything stays in its cache, but since the operations at the *clients* are a message send followed by a receive, each client is no longer stalled waiting for the writes to complete: they can context switch to another thread and do something else useful. This extra time is more than 5000 cycles in the extreme case. You can do a lot in 5000 cycles.

## 8.7 Memory consistency

Implementing correct communication in UMP is not quite as easy as it sounds, particularly on ARM machines. It is important to pay attention to the *memory consistency model*. Recall that a system's consistency model refers when CPU cores observe a change to memory when another core modifies it.

Sequential consistency implies that all writes to memory are totally ordered, and observed as such by every core. Sequential consistency is easy to reason about, but very hard to implement efficiently. Consequently, almost no real multiprocessors implement sequential consistency.

Intel architecture machines implement a weaker form for consistency called Total Store Ordering (TSO), which is roughly equivalent to Processor Consistency: writes from one core are seen in the same order by every core, but each core may observe different interleavings of writes from different cores.

As we have described UMP so far, as long as the sending processor issues the writes to the cache line in program order (that is, the order in which they appear in the code), the channel is correct on an Intel architecture machine using TSO.

Unfortunately, ARM processors implement an extremely weak form of memory con-



Technical  
Details

sistency, a fact which caused Barreelfish to break rather badly when first ported to multicore ARM systems (including the one you are working on).

On an ARMv7-A processor, barriers need to be used to ensure the required orderings of memory operations. The ARM memory model has different consistency rules for “synchronizing accesses” (such as atomic RMW operations): such accesses are sequentially consistent, and also act as a barrier: any writes in program order will complete before the synchronizing access, and all reads and writes following it in program order will not start until the access has completed.

The moral is that it is essential to know your hardware, and in particular, the consistency model it implements.

ARMv7-A provides three different explicit barrier instruction:

- `dmb` is a data memory barrier: stores before this point complete before loads and stores that come afterward, in program order.
- `isb` is an instruction synchronization barrier. Instructions beyond this point in program order are not fetched until this barrier completes.
- `dsb` is a data synchronization barrier. It is a stricter form of `dmb` that takes into account other architectural events, and need not concern us here.

Both `dmb` and `dsb` actually come in different flavors which change the scope and subtype of barrier that they implement, but in this course we'll only be concerned with a full data barrier that applies system-wide to loads and stores.

For example, consider the problem of self-modifying code: writing an opcode (held in register `x0`) to a memory location (whose address is in register `x1`), and then jumping to that instruction (such as might happen when removing breakpoints in a debugger). The correct sequence of barriers is as follows:

```

1 str x0, [x1] // Store opcode to *x1
2 dmb      // Barrier
3 isb      // Barrier
4 b x1    // Branch to *x1

```

We need two barriers here. The `dmb` ensures that the store to `*x1` completes before `dmb` itself completes, which itself completes before the branch. However, the `isb` is also required, to ensure that the branch target isn't fetched until the `dmb` has completed.

As you can see, the use of barriers can be subtle, and is a frequently source of tricky concurrency bugs.

What does this mean for UMP? Consider the following operations at the sender and receiver of a UMP channel; we've simplified the cache line into simply a single data word and a flag indicating valid data:

Sender	Receiver
data = x;	while(flag);!
flag = 1;	y = data;

This code doesn't work. A pair of ARM processors is perfectly capable of executing these four operations in the following order:

Sender	Receiver
4: data = x;	2: while(flag);!
1: flag = 1;	3: y = data;

You might think that putting in a barrier to force the sender to write the flag after writing the data would fix the problem:

Sender	Receiver
data = x;	while(flag);!
dmb;	y = data;
flag = 1;	

You'd be wrong. The following observable execution order is valid on ARM, and really happens (note that there are no dependencies between the two read operations at the receiver):

Sender	Receiver
2: data = x;	5: while(flag);!
3: dmb;	1: y = data;
4: flag = 1;	

You need barriers on both sides. The following is correct UMP code (phew!):

Sender	Receiver
data = x;	while(flag);!
dmb;	dmb;
flag = 1;	y = data;

If you find this too confusing, you're not alone. In the subsequent (64-bit) version of the ARM architecture, ARMv8, ARM Ltd. decided to make the consistency model stricter (at the cost of more complex hardware) to make it less likely that programmers would make errors like this. It was probably a wise move: a quick audit of the Linux kernel source last year found many cases where barriers were missing or, sometimes, inserted where they did nothing useful.

## 8.8 Getting prepared

In this milestone, you'll implement a form a UMP for your OS. You'll implement unidirectional communication channels, each of which is a fixed-size circular buffer,



Project  
Instructions

and where all messages are 64-byte cache lines (cache lines on ARMv7-A are, actually, 32 bytes wide, but here we'll use them in pairs).

You'll be extending the RPC interface you already have using LMP to work between cores using a simple UMP-style channel. You don't have to worry too much (yet) about optimal performance, but you *do* need to worry about the caches and ARM's memory model.

If you do not have working user-level threads already, it may be helpful to get them working at this point in order to make some of the implementation for this milestone easier (e.g. polling on the cross-core channel).



*Project  
Instructions*

## 8.9 Sharing a Frame Between Cores

The first thing we need is some shared memory. Recall the URPC frame that you used in milestone 5, to pass RAM parameters to the second core. On core 1, this frame is available in the task CNode. Make sure that you map this frame cacheable – using an uncached mapping is fine for testing, but it's cheating in the final submission (it's *really* slow).

Also, pay attention to ARM's weak memory model. You can't count on your stores being observed by the other core in the same order that you write them down, so you will need to insert barriers as appropriate to ensure that your protocol is correct. It's important as well that the compiler doesn't reorder your instructions; you should definitely look at the assembly output in this case to make sure it's all OK.

Part of the grading for this milestone is being able to explain your implementation and demonstrate that you understand why it is correct.



*Project  
Instructions*

## 8.10 Establishing a communication protocol

Once you have shared memory between the two `init` processes, you should establish a protocol which enables meaningful communication between them. As our requirements from this communication channel are modest, a simple UMP-like protocol should suffice.

Our ultimate goal is to run only one shell which will be responsible for spawning applications on both cores. For this, we only initially need primary-secondary communications. As the primary application can wait while a process is starting on the remote core, we can implement this communication as RPC.

Essentially, we need to support a *remote spawn* message as well as a corresponding response on this channel. In this remote spawn message, `init.0` should send the name of an application to `init.1` which should then start this application and report the status (success/failure) to `init.0`. The shell on the BSP core can send requests for *remote spawn* over LMP to `init.0` which then forwards the message

to `init.1` (using our new shared-memory communication channel) for actual execution. Similarly, responses can go back from `init.1` to the shell on the BSP core.

To make this work with your existing RPC interface – which might not have considered the fact that you might want to spawn an application on another core – you have to make sure that your existing `aos_rpc_process_spawn` RPC correctly handles the `core` argument.

```
1 errval_t aos_rpc_process_spawn(struct aos_rpc *chan, char *name,
2                                coreid_t core, domainid_t *newpid);
```

You will also need to take the necessary steps inside your process management system to enable spawning processes on the second core over a channel like the one described above.

Once you're able to send messages between cores, you're ready to start launching processes. Remember to make sure that you've correctly mapped the `bootinfo` structure, where the ELF images live, into your address space on core 1.

You should now extend your UMP protocol, to allow all RPC operations to be forwarded between cores. A sufficient implementation is to route all RPC calls via the `init` process (or equivalent), which forwards them on behalf of user-level applications.

You must be able to reach any RPC server on core 0, from an application on core 1, and vice versa. You should provide an interface for applications to *bind* to servers, and then use this binding to make RPCs.

Note that you need to make sure that you don't accidentally forward messages that should be handled by an application on the same core.

## 8.11 Fragmentation and reassembly

Cache line sizes vary from processor to processor. On ARMv7-A and older 32-bit x86 processors they are generally 32 bytes, whereas on ARMv8-A and 64-bit x86 processors the are 64 bytes in size, although Marvell Cavium ThunderX-1 cores have 128-byte cache lines.

If you reserve the last word of a cache line for flags, that leaves either 28 byte or 56 bytes for each message. This may work for spawning a process, but is not enough for passing command line arguments, for example. To get this work, you need to split a single application-level message into multiple successive UMP messages (sometimes called *fragmentation*) and then *reassemble* them at the other end.

The first extra challenge is to do precisely this: demonstrate that you can pass fairly large messages as a sequence of small, fixed-size UMP slots.



Extra  
Challenge

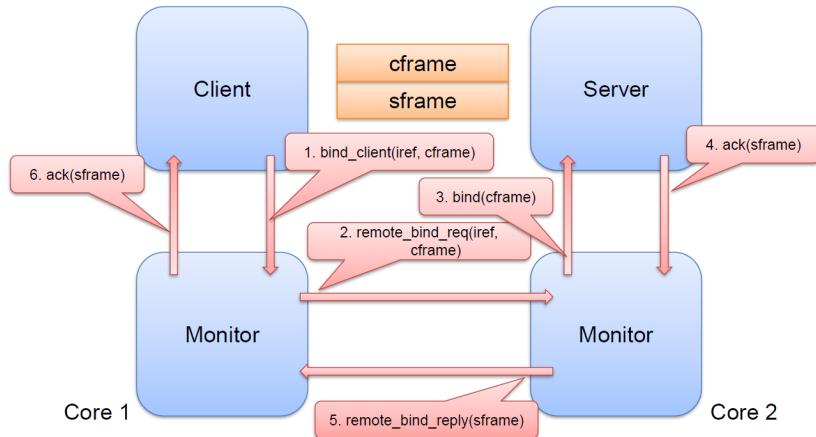


Figure 8.11: The Binding process



## 8.12 Performance

While optimal performance is not a requirement for this milestone, it's a good extra challenge to optimize it as much as possible (much as it was with LRPC). However, the first step in doing this is actually measuring the performance of your implementation.

The first extra challenge is therefore benchmarking: show detailed graphs showing the performance of your UMP implementation, and breakdowns of where the time goes in the system.

Once you've got this data, and you can explain it, you then have a basis for claiming that your optimizations actually do improve performance.



## 8.13 Binding

So far, you are using a single communication channel between the two cores. Obviously, it can be useful to be able to establish direct communication between processes on any pair of cores in a system.

To fully implement any-to-any communication, the process of *binding* a client to a server has to be considerably more complex than in this milestone so far. It turns out that in scenarios in networking or distributed systems, the business of establishing and then tearing down bindings is usually much more complex, and involves much more code, than the "data plane" operations of simply sending and receiving data along a channel.

Binding in Barreelfish is no exception. Figure 8.11 shows the sequence of operations in regular Barreelfish; the goal is to end up with a couple of shared memory regions

(one for each direction of communication).

It's often useful to have a third-party helper for setting up connections: the client generally doesn't know how to contact the server, but it knows how to contact something else that knows. In this case, the monitors on each core act as *brokers*, matching a client request on one core to a server offer on another core.

The process of binding proceeds as follows:

1. The client allocates and maps a region of shared memory (the cframe in Figure 8.11).
2. The client calls its local monitor with a capability to this frame, and the identifier of the server it wants to connect to.
3. The client's monitor figures out which monitor is on the same core as the server, and forwards the request to it.
4. The server's monitor calls the server, giving it the client's cframe.
5. Assuming the server decides to accept the connection, it allocates another region (the sframe) and returns this back to its local monitor
6. The server's monitor returns the sframe capability back to the client via the client's local monitor.
7. The client maps the server's region, and both sides are now ready to go.

Quite a lot needs to happen for this work. Firstly, you need to be able to transfer capabilities between cores. Then, you need a way to name each server that might be offering a service. The monitors need to keep a catalog of where each named service can be found. This implies another process (which we haven't talked about) that allows a server process to tell its local monitor that it has a service to offer.

## 8.14 Arbitrary channels

If you like, an extra challenge is to implement this generic communication between processes across cores so that any two processes can setup a direct channel with each other. This will require a more sophisticated binding operation, as the processes will need to share memory.

You will also need to figure out how to *name* communication end-points in processes, so that one process can tell the system what it wants to connect to.

This can be quite a bit of work, but on the other hand it's a useful experience, and it will also make it much easier to integrate some of the individual projects in the final part of the course.



Extra  
Challenge



Commentary

## 8.15 How to do well

This is the last milestone before the individual projects, and it will save a lot of effort later on if this is done well.

Probably the most tricky part of this assignment is the correct use of barriers (and being able to explain why). The protocol is a little more complex than the description we've given here: in particular, we haven't shown how to recycle slots in the ring buffer.

The system also now has two different protocols for communication: LMP and UMP. You'll need a story for how to listen on both of these at the same time, and be able to explain it.



*Project  
Instructions*

## 8.16 Milestone 6 summary

You will be expected to demonstrate the following functionality:

- You can start applications that run on the second core from a program running on the BSP core.
- Processes on the second core can access RPC services on the first core e.g. memory server, or serial console.

In addition, you are expected to be able to explain the following aspects of your work:

- How your user-level messaging protocol works in sending and receiving messages.
- How you set up the channel to enable communication.
- Where you put barrier instructions, and why.

Extra challenges:

- Sending of large messages across cores.
- Measure and demonstrate the performance of your communication implementation.
- A full binding system for arbitrary process-to-process communication

## **Part III**

# **Individual projects: building on the core**



By now your team has the foundation of a multikernel operating system working on your platform: processes, threads, virtual memory, inter-process communication, and basic I/O.

The rest of the course is about developing a set of modules which run over this OS and add more functionality. Each team member should choose one of the following projects and work on it, but it should be clear that they fit together fairly naturally, and so it's important to work closely with your teammates.

We strongly urge that somebody in the team takes on the Shell project (described first). At the end of the course you will have to present an integrated demonstration of the complete system, and this is much more compelling with a shell. A shell is also an invaluable aid for other projects, since it can provide an environment for testing, debugging, and automation. It's also good for morale: a shell really makes a system feel like it *works*.

Integration is an important factor in how we award points for the final milestones. Make sure that your various individual projects work together, and that you can demonstrate them on the same machine with the same image – ideally without rebooting in between.

Finally, as with the course milestones so far, write up the final report (Chapter 14) as you go along. Remember, there is no page limit for the report, and the more interesting information you can put into it the better.



# Chapter 9

## A Shell

In this project, you'll develop a command-line interface (shell) with which you can demo and coordinate both the existing OS functionality that your team has built, and the other individual projects your fellow team members are working on.

There are really two aspects to this project, which are interleaved. The first is the system infrastructure required to run the shell:

- Running the UART driver in user-space rather than inside the CPU driver.
  - Reading from the console port as well as writing to it.
  - The ability to use the console driver from other processes, so any process can implement a command line.
- and optionally:
- Logging into the machine over the network, if one of your team is doing the network stack project.

The second aspect is the core functionality of the shell itself:

- Basic reading a line, parsing, and executing it
  - A set of useful built-in commands
  - The ability to run programs from the multiboot image
- and optionally:
- If you have a teammate who is working on file system support, the ability to redirect I/O to and from files and load and run programs from the file system.
  - Inter-process communication set up by the shell – essentially the equivalent of pipes.

We start with the basic stuff...



## 9.1 Keyboard input

In milestone 3 you implemented a terminal service which allows any process to write characters to the console. For this project, you're also going to need to provide these other processes with serial input as well, so that they can request input from the user. We'll build this functionality up in stages, starting with being able to read a character from the UART by accessing it in userspace. You can find a complete overview of what the i.MX8X LPUART can do in *Section 13.6* of the i.MX 8DualXPlus/8QuadXPlus Applications Processor Reference Manual.

If you want to use some of the more advanced functionality, we recommend looking at the fairly complete Mackerel device definition for the LPUART in `devices/lpuart.dev`.

### A userspace UART driver

Drivers in Barreelfish run in userspace. The first step in setting up a userspace driver is to make the registers of the device it should control available to it. There is a special capability type to refer to memory mapped device registers in Barreelfish: a `DevFrame`.

When `init` is started, it receives a cap in its `TASKCN_SLOT_DEV` which contains most register regions for the devices on the Toradex board. Your job is to manage this region and retype it to smaller capabilities of type `ObjType_DevFrame` according to the sizes and base addresses of register regions for individual devices. This will also be required for other projects so talk to your teammates! If you decide to run the driver for the LPUART within `init` you simply need to map the device frame with its registers into `init`'s virtual address space.

A more advanced way would be to have a separate domain for the driver. Such a domain would somehow need to get hold of the appropriate device frame. How you hand out the device register regions is up to you. In mainline Barreelfish, drivers used to request the device frame capabilities from the device manager via RPCs. Drivers being able to request access to arbitrary device regions of course has some security implications. This is why the resources are now handed to drivers by putting the required capabilities into a dedicated CNode (the ARGCN) when spawning them. You should have most of the infrastructure already in place for spawning a process with a capability handed over from the process starting it, but you would probably need slight modifications to your spawning code.

We provide a driver for the LPUART in `usr/drivers/lpuart/`. To use the library, include the header file `include/drivers/lpuart.h` and add the `lpuart` library as a dependency (in `addLibraries` in the `Hakefile`). The initialization function takes a virtual base address at which it expects the registers to be accessible. The header file `include/maps/imx8x_map.h` contains the base addresses and sizes of the register regions for the four UARTS on the Toradex board (and the ones for some other devices important for this course). The serial-to-USB converter is wired to LPUART3, so you should use this one for printing and reading from your default

console. Make sure to map its register region read/write without caching enabled.

### Basic character input

After you have successfully called the `lpuart_init` function, you should be able to print characters using `lpuart_putchar`. You can also read using `lpuart_getchar`, this function is non-blocking, if there is nothing to read, it will return `LPUART_ERR_NO_DATA`. But how do you know when you should read from the UART? You could poll the UART, but this would block the rest of your system. A better way is to use an interrupt to get notified when there is data ready. We can tell the UART to start generating receive interrupts by using `lpuart_enable_interrupt`.

But where do these interrupts end up? In fact, only telling the UART to generate interrupts is not enough, we also have to setup the interrupt forwarding in the interrupt controller, and attach a handler to the interrupt. The CPU driver does only interact in a minimal way with the interrupt controller: It enables the timer interrupt for internal use, acknowledges and forwards other interrupts to userspace. It does not, however, configure generic device interrupts. We have to do this in userspace as well. We provide a driver in `usr/drivers/gic_dist` that can configure the GIC distributor. It also requires the GIC to be mapped in the current address space. The physical location of the GIC distributor on the i.MX8X can again be found in the header file `include/maps/imx8x_map.h`. As with the UART registers, it is important to map this region read/write with caching disabled.

By now, the interrupts are able to reach the CPU driver. But we still need to tell the CPU driver to deliver those interrupts to our domain. There are multiple capabilities involved: First `IRQTable`, which represents a whole interrupt vector table, and `IRQDest` which represents a set of entries in such a table. You can obtain `IRQDest` from the `IRQTable`, finally you can connect a single vector in a `IRQDest` to an LMP endpoint, and you will receive a message on that endpoint if an interrupt has occurred. We provide two functions code in `lib-aos/inthandler.c` (respectively `include-aos/inthandler.h`). Unlike on x86, on ARM the vector that is triggered is given by the hardware. Hence, when you obtain the destination capability, it is important to pass the interrupt vector number of the UART. The header file `include/drivers/lpuart.h` contains definitions for the interrupts generated by the UARTS.

It is important to setup the forwarding and attach the handler before enabling the device interrupts: Otherwise, an interrupt might reach the cpu driver while not being able to handle it. Ideally you follow the following sequence for enabling interrupts:

- Initialize the GIC distributor driver
- Initialize the LPUART driver
- Obtain the IRQ destination cap and attach a handler to it
- Enable the interrupt in the GIC distributor

- Enable the interrupt in the LPUART

## Using the user-space serial driver from other processes

We only have one serial port available. You will now need to think of a way to multiplex this resource such that the characters read from the port always go to the correct domain – typically, the one that is currently using the serial port.

It's best to do this on a line-by-line basis. This means that you will have to buffer the characters from the serial line until you see an newline (ASCII NL, 10), carriage return (CR, 13), or end-of-file (EOT, 4) character.

Once your process can read and buffer characters, the next step is to make sure that any process can use this new serial driver. For this, we will add a new RPC call that allows any process to read characters from the serial driver. The RPC call is listed below, and the skeleton is provided in the code handout (we'll test your code against this):

```
1 char aos_rpc_serial_getchar()
```

Any application should be able to use these calls for input/output of characters.

## Use RPC calls for libc functions

In order to use your own terminal read functions from any process you need to set the libc function pointers accordingly. Change the lines in `barrelfish_libc_glue_init` to point to your new read functions.

```
lib-aos/init.c
1 void barrelfish_libc_glue_init(void)
2 {
3
4     _libc_terminal_read_func = aos_terminal_read;
5     /* Already implemented */
6     _libc_terminal_write_func = aos_terminal_write;
7     /* ... */
8 }
```

By this point, any process should be able to read and write line-oriented characters from the serial port.



*Project  
Instructions*

## 9.2 Implement a command-line interface

It's now time to write the shell itself, as a small command-line interface with a set of built-in commands. The shell reads a line from input, parses it into tokens, and then does something based on the first token (including printing output).

Here are some examples of functionality (built-in commands) that such a command-line interface could provide:

- echo which takes a string of characters and echoes them back
- turn on/off LED
- A demo of working user-level threads
- run\_memtest which takes the size of the memory region to test as an argument and runs the memory test in a user-level thread.
- oncore run an application on a certain core
- ps showing the currently running processes
- help show the available commands
- ls, cat, mkdir, rmdir if you implemented the filesystem
- A time command, which measures the time taken to execute another command.
- Any other functionality that you might find in a shell, such as Super Cow powers.

## 9.3 Process Management

Builtins are essential for a shell, but most command line interpreters derive most of their power by being able to run other programs and connect them together.

The next step is to be able to create new processes from the shell and run them. If your teammates don't have a file system working yet, you can run images from the multiboot image.

There are two options for how to do this. Some shells have a separate `run` command which takes the name of a program as an argument. More usual, however, though somewhat less secure, is the Unix-style of looking for programs in the file system when the user supplies a command name which is not a builtin.

Besides starting new processes from the shell (on either core), you should also be able to list (`ps`) currently running processes, and have a way to terminate (`kill`) them. This will require some way of communicating that the process has terminated – a functionality that is required when you start a new process in the foreground. Note, maybe offering the ability to terminate, say, `init` may leave your system in an instable state, so maybe it's better to prohibit this.



Project  
Instructions

## 9.4 Implement I/O redirection and pipes

Ideally, the shell should be able to start a new process while redirecting its standard input and output to or from a file. Again, you need a teammate to have implemented a file system to really show off this functionality, but you can have it read from a datafile inserted in the multiboot image.

For an even more impressive challenge, implement pipes: redirect the standard output of one program into the standard input of another, and add the “|” operator to the shell to make this happen.



Extra  
Challenge



Extra  
Challenge

## 9.5 Network login

If someone else in your team is building the network stack, this is a great extra challenge, but be warned that it can be a lot of work. The idea is very simple: from a Unix machine, you should be able to type:

```
$ telnet 192.168.0.2
```

– or whatever network address the machine has, and get a shell. This is going to need the network stack to support some form of TCP, though you might be able to hack together a simple protocol based on UDP if TCP is not available.

One reason this is more work than it seems is that you need a server (the equivalent of `telnetd` or `sshd` which is listening on a socket, and creating shells on demand).

The full complexity of this in Unix is called the “TTY subsystem”, and is one of the most complex, and least understood, parts of the Unix kernel.



Commentary

## 9.6 How to do well

Is your I/O library thread-safe? Can you demonstrate this?

The way to do really well in this individual project is essentially to deliver a useful shell, and your teammates will appreciate this as well.

You may be tempted to implement useful shell user-interface features, like history, command editing, and completion to your shell. By all means do this, but it's a bad idea to let this distract you from core functionality (or, indeed, some of the extra challenges). If you *do* have the time, it might be better to borrow and port the readline code from the FreeBSD libraries, for example, rather than writing it yourself.



Project  
Instructions

## 9.7 Summary

You are expected to demonstrate the following functionality:

- A working userspace driver for the UART
- Your UART driver is able to accept input and print output from user space.
- Any application is able to use the UART driver for standard input and output.
- A command line interface that can demonstrate other parts of your system

Extra challenges:

- I/O redirection and pipes
- Network login



# Chapter 10

## Filesystem

In this project you will implement a file system. We'll give you a block driver for the SD card reader, you will write a filesystem above this driver. After that, you will use your file system to read ELF binaries and be able to spawn them as processes inside your system.

The work consists of:

- Getting a block driver working
- Measuring its performance
- Implementing a file system on top of the block device
- Handle reading of files and directories
- Handle writing of files and directories
- Integration of your filesystem implementation into the standard functions such as `fopen` and `fread`

### 10.1 Setting up the block driver

The i.MX8X contains three uSDHC interfaces, on your Toradex board, one is hard-wired to the onboard eMMC flash, one is connected to the external SD card reader and the last one is unused. The one you should use is the one connected to the SD card reader. It is referred to in the datasheet as SDHC2.

We provide you with a library (`/usr/drivers/shdc/`) for the block device driver. This library expects the virtual address of one of the SDHC registers, hence you have to map the registers first and then pass the virtual address to the library. It's up to you to determine where to put the block driver, but the access to the controller has to be synchronized somehow. One option to achieve this is to have a separate



Project  
Instructions

domain running that talks to the SD card and offers a block level interface to clients, then run the rest of the file system implementation in the clients. To use the library, include the header file `include/drivers/sdhc.h` and add the `sdhc` library as a dependency. Refer to the description in [section 9.1](#) for more details on how to set up a userspace driver.

The driver library is somewhat limited: It expects to talk to a SD high-capacity card. Hence you need to use a card in the 4 to 32 GB range, ideally the one we provide you with. This limitation also has a bright side: If you accidentally point the driver to the controller connected to the internal flash, it won't work. So you don't have to worry about bricking your device. The driver is also fairly slow, it does not do any negotiation with the card to find ideal working parameters (such as bit-width, clock frequency, etc.), but always runs in the most compatible settings.



*Project  
Instructions*

## 10.2 Testing the block driver

Traditionally, filesystems build on top of a block device driver that is able to read and write fixed-size units of data (usually 512 bytes), called *blocks*, to and from a disk, flash drive, SD card or another storage medium. The block driver we give you for the SD card reader has such a simple interface to read and write one block at the time:

`include/drivers/sdhc.h`

```

1 errval_t sdhc_write_block(struct sdhc_s* sd, int index, laddr_t source);
2 errval_t sdhc_read_block(struct sdhc_s* sd, int index, laddr_t dest);
```

These two functions read or write blocks of 512 bytes in size to and from the SD card. The `source` (`dest`) argument points to the physical address of a pre-allocated buffer which is at least 512 bytes in size. The driver uses DMA internally, the controller will directly read/write the transferred data from memory. The library will not do any cache flushes/validation for you, hence you have to make sure that before calling the write function, the cache is clean and after a read that you invalidate the virtual address range of the buffer. This is enough to get a basic file system working, although it won't be fast.

You can show how fast (or otherwise) it is by benchmarking reads from, and writes to, an SD card, and including the results and graphs in your report. Ideally, you should provide a breakdown of where the time goes in this implementation: waiting for the device, communication overhead, delays because of the synchronous one-block-at-a-time interface, etc.



*Extra  
Challenge*

## 10.3 A faster block driver

Unfortunately, the block device driver you'll get is very simple and slow. Fortunately, during this course, you have now mastered all the necessary skills to write a fast

device driver. We will award bonus points for any improvements you can do in the driver. This includes for example reading/writing more than one block at the time, using interrupts to transfer data to and from the card instead of polling, or configure the host controller to increase the clock frequency of the card. Be sure to measure the performance improvements you get from your changes and show them to us in the presentation.

You can find complete overview of what the i.MX8x SDHC can do in *Section 14.8* of the i.MX 8DualXPlus/8QuadXPlus Applications Processor Reference Manual. We already found one inconsistency in the datasheet: The VEND\_SPEC register has additional bits that have to be set. So be aware, also datasheets can have bugs or be incomplete.

If you choose to take on this extra challenge, you should supply further performance analysis in the report to show that your optimizations actually work, or not (we're always happy to see negative results as well as positive ones).

## 10.4 The filesystem itself

Your next task is to write a filesystem for the block device driver. We strongly suggest you implement a filesystem that can read a FAT32 partitioned SD card: this is easy to implement, and also test, on a Linux machine to check that you're doing the right thing in your OS.

You find a link to the official FAT32 specification from Microsoft on the course website. FAT is a fairly simple filesystem (the specification is only 34 pages!) and therefore well-understood and well-documented. It's not the greatest design of file system, however, but that's beyond the scope of this course (at the moment).

In order to test your filesystem, you need to create a FAT32 file system on the SD cards we handed out to you. On a Linux system, you do that by issuing the following command (where /dev/sdX is the device node corresponding to the SD card - make sure you don't reformat your root file system by mistake!):

```
$ mkfs.vfat -I -F 32 -S 512 -s 8 /dev/sdX
```

This command creates a FAT image on the SD card with a logical sector size of 512 bytes and 8 sectors per cluster. Make sure you use the right device from /dev when formatting! You can use dmesg to figure out which device was recently mounted. Once you have partitioned the card you can start the provided mmchs driver from the hand-out. If everything worked correctly, the driver will print the contents of the first sector. You can easily verify that you're dealing with a FAT disk in case the first byte on the block is 235 (0xEB) and the third byte is 144 (0x90).

You can assume that there will be just one partition on the SD card.

If you don't have an SD card reader you can use the Toradex board as one: Reset it and press Ctrl-c once U-Boot is waiting for an image to be loaded over USB.



Project  
Instructions

This drops you into a shell where you can type the following command to expose the SD card as a mass storage device over the USB OTG port:

```
=> ums 0 mmc 1:0
```

FAT filenames are restricted to 11 characters without long-filename support (VFAT). You can just use small direntries in your filesystem implementation. Note, that the filename has to be unique and UPPERCASE.



*Extra  
Challenge*

## 10.5 Add the support for long file entries

If you like, you can add support for long file entries to your basic implementation. This is not too difficult, but bear in mind that long filenames in the Windows world don't encode characters in UTF-8 format (the de facto standard for Unix) but UTF-16, itself an extension of UCS-16.



*Commentary*

## 10.6 Design Space

As already stated, we give you freedom on how you want to implement the support for file system. In the end you will need to provide the functionality for any domain in your system to call `fopen` and `fread` for instance (see the setup below). There are multiple ways to implement this functionality. Make sure you will explain and describe your design choice in the final report with enough detail. In any case, you will need to serialize the access to the SD card device to avoid race conditions when accessing blocks. For each variant there will be a different set of RPCs you will need to implement. For instance, if you decide to implement a file service you need to provide different RPC calls than if you decide to implement the filesystem as a library. In the latter case you just export block read and write functions for instance. Make sure you also explain the RPCs you've added.



*Project  
Instructions*

## 10.7 Implementing a file system

You will need to tell the C library how to open files and directores for reading and also writing to them. Before you can use the file system you are supposed to call the function `filesystem_init()` which initializes a dummy filesystem and then calls the following function:

```
+ void fs_libc_init(void *fs_state);
```

You will see that this function sets function pointers to various file and directory related functions used by the C library. Currently, it configures the C library to talk to a simple ramfs implemented as a library. When you imlement your file system, you will need to replace the initialization with your FAT32 code and also set the function pointers in `fs_libc_init` accordingly.

This will set up the function pointers to be used by the C library accordingly so you can start opening and reading files and list contents of directories. You may have a look at `/usr/test/filereader` to see a sample programm that should be supported by your filesystem.

### 10.7.1 Read support

Next, you should implement the basic support for a read-only filesystem. This is *much* simpler than a writeable file system – you don't have to worry about consistency, durability, atomic updates, space allocation, integrity checking, and whole host of other issues.

It does include operations such as:

- Opening of existing files: `fopen` and `fclose`
- Reading file contents: `fread` and friends
- Positioning the cursor: `ftell`, `fseek` and friends
- Listing contents of directories: `readdir`, etc.

### 10.7.2 Write support

Once you've got this working, your basic implementation should be able to deal with reading existing files and directories. Here is a good stage at which to release it to your teammates: those working on the other projects will probably appreciate even a rudimentary read-only file system, and will also help you find bugs!

The next step is to provide basic support for writing to files and directories:

- Opening (creation) of non-existing files in `fopen`
- Writing file contents: `fwrite` and friends
- Truncating files: `ftruncate`
- Creating new directories: `mkdir`
- Removing files and (non-empty) directories.

This will require allocating new unused blocks/clusters and freeing up unused ones. You will also need to be able to extend existing cluster chains with the newly allocated block in order to grow files.

FAT32 is fairly simple in this regard: the File Allocation Table which gives the file system its name holds information on all the existing clusters in the system. An entry is either free (0x000 0000), points to the next block in the chain, or marks an end (0xffff fff8 / 0xffff ffff). Note the top 4 bits are reserved in the entries.

### 10.7.3 ELF loading

Now that your filesystem is working, you are able to add ELF loading support to your OS.

Until now, your build system assembled a multiboot compliant boot image at compile time which contained a set of modules (i.e., the ones you specified in your menu.lst file inside your build directory), those modules are already just regular ELF files, therefore you can think of it as an archive of ELF files. In a real system we do not want to load only binaries that ship with the kernel. Using your filesystem, you can now add support to load ELF files from an SD card. That involves loading an ELF file from your card into memory, then doing necessary ELF relocation and setting up the address space of your newly allocated process in a way that it can execute the binary. Note that how you implement this part and also how much work you have to do for this assignment depends a lot on how your system looks like at this point (especially the process management, your file and memory management subsystems).

We will provide you with code that already does much of the ELF handling for you (have a look at the ELF library in lib/elf). For an example of how to use the library, you can have a look at how the kernel code loads the init process from the multiboot image.

Note that if you have a fellow team member who is implementing the shell (Chapter 9), it would be good to coordinate over this.



*Project  
Instructions*

## 10.8 Performance Evaluation

As a last step you'll have to evaluate the read performance of your filesystem. Your evaluation should at least contain the maximum read bandwidth you achieved. You need to be able to explain your results, where the potential bottlenecks are in your implementation and how you could improve them. You might find the timer functions in aos/systime.h helpful.



*Extra  
Challenge*

## 10.9 VFS, MBFS, NFS, ...

Unix-like systems frequently implement a layer in the kernel between the generic file system API (provided by libc) and the backend implementation of the file system [65], and regular Barrelnfish adopts this interface as well.

Among other advantages, the VFS layer allows different file system implementations to be combined into a single name space using file system mounting. For example, you can have the root filesystem pointing to the ramfs library and create a folder e.g. /mnt/sdcard where you mount the SD-card with your FAT32 file system.

As well as the filesystem you have implemented, there is at least one other implementation readily to hand: the list of modules in the OS's multiboot image. A first

step after implementing a VFS layer between your library interface and FAT32 is to add a VFS interface to this (read-only) file system, and mount the latter in the same name space.

A bigger prize (and more work), assuming one of your fellow team members is working on the Network Stack project (Chapter 11), is to implement a simple NFS client using UDP, and demonstrate mounting an NFS volume over the network. This is, in many ways, much easier than the FAT32 implementation since it's basically an RPC protocol, but it gives your OS access to networked files.



Commentary

## 10.10 How to do well

As with the other individual projects, integration is the key here. For this reason, it's a really good idea to get your fellow team members using your code as early as possible – as soon as you have a workable read-only file system for example.

Conversely, you'll find it much easier to test (and automate testing for!) your filesystem if you have access to an early version of a shell written by someone else in your team.

Pay attention to corner cases in the code, and be prepared to explain them. For example, what happens if you delete a file and there are still clients that have a file descriptor to that particular file?

Finally, as with many of the other projects, be prepared to explain the performance results you obtain. Being able to explain them is much more important than making them fast!



Project  
Instructions

## 10.11 Summary

You are expected to demonstrate the following functionality:

- Reading files in a read-only SD card file system, demonstrating `ls` and `cat` commands
- Writing files and creating directories, as well as deleting them.
- Loading ELF binaries from the file system and running them.
- Performance analysis of the block driver
- Performance analysis of the file system itself.

Extra challenges:

- Optimizing the performance of the block driver
- Adding support for FAT32 long filenames
- Virtual file system support, possibly including an NFS client

# Chapter 11

## Networking

In this project, you will develop a simple network stack for your operating system. It will allow applications to communicate with the outside world using the UDP protocol (and, if you're ambitious, TCP).

The work consists of:

- Getting the registers for the driver.
- Understanding how to use the device queue interface to send/receive raw packets.
- Implementing ARP, Ethernet.
- Implementing IP, UDP, and ICMP.
- Handling multiple network clients in different processes.
- Measuring the performance of the implementation.

Extra challenges include:

- Simple remote command line over UDP
- Basic TCP support

### 11.1 Overview

A full network stack is a big and complex thing, including protocol implementations and drivers for network adaptors. This project is deliberately minimalist in its approach, though there are plenty of additional challenges if you feel ambitious. We make the following initial simplifications:

- The goal is to demonstrate UDP and ICMP functionality.



Commentary

- You don't need to write a network device driver. We'll provide a working driver for the 100 Mbit ENET (documented in chapter 14.6 [84]).

However, we do require you to provide network connectivity to multiple applications, which means multiplexing and demultiplexing packets for different endpoints.



*Technical  
Details*

## 11.2 Network hardware

The Toradex board features a 100 Mbit Ethernet port with a Medium Dependant Interface (MDI). The Transceiver base on the Micrel KSZ8041 (PHY) is connected over RMII (Reduced Media Independend Interface) with the MAC contained on the SoC. The MAC on the SOC is called Ethernet MAC (ENET) and the control of this hardware is directly exposed through registers. On the other hand the PHY is only indirectly programmable through the RMII protocol which is exposed to the programmer by the register ENET\_MMFR (14.6.5.6).

Even though the Ethernet MAC is not a high end network device, it already has quite a few features:

- Packet checksum generation and checking
- Frame padding (if your packet is less than 64 bytes, it will be padded to 64 bytes since other network stacks might drop packets less than 64 bytes)
- VLAN-tagged frames according to IEEE 802.1Q [73]
- Programmable promiscuous mode to omit MAC destination address checking on receive
- Multicast and unicast address filtering
- Statistics indicators fro frame traffic and errors
- Interrupt coalescing
- AVB (Audio Video Bridging)
- Multiple hardware receive/send descriptor queues

All of these features are programmable through either a specific register or a set of registers. A Table of all the registers is available in Section 14.6.5.

The provided driver (`usr/drivers/enet`) is rather simplistic. It only uses a single receive and send queue and does not implement the check sum generation. Furthermore, we do not handle interrupts i.e. you will have to poll the send/receive queues. Feel free to modify the driver and enable new features that might make your life easier.

Make sure that you connect a cable to the Ethernet port (and attach it to another computer), otherwise the driver will not complete it's initialization. The reason for this is the autonegotiation of features and speed of the link. Without this feature, the link layer will not finish the initialization and fail.

In the handout we already encapsulated the send and receive queue into a struct which is usable with our device queue interface (`include/devif/queue_interface.h`). In the driver domain, the main function of the driver (`usr/drivers/enet/enet_module.c`) is polling the receive queue and already receives unprocessed raw packets.



### 11.3 Starting the driver

We provided you with the code for the driver, there is only a single part missing: you need to get the register regions for the device. The registers of each device have a specific address and size. The definitions of these two values for the ENET device is defined in `include/maps/imx8x_map.h` as are many others. You can find more details about how to give the driver access to the necessary resources in [section 9.1](#).



### 11.4 Sending/Receiving raw packets

The simplest way to implement the network stack is directly in the process of the driver and expose it as a service to other processes by extending your RPC implementation. This is just a suggestion and you are free to come up with your own design.

The first task is to understand how to interact with the hardware send and receive descriptor queues. The queues are abstracted using our device queue framework also present in Barrelyfish. In this section we will only give you a short introduction on the interface, for more information you can read the Barrelyfish Technote No. 26<sup>1</sup>

The framework is based on the simple idea of transferring ownership of buffers. When a process allocates memory, that process owns the memory and is free to modify it. In the case of our network driver and the hardware device, the driver (or some other process) can allocate memory and then transfer the ownership of parts of that memory to the device. (i.e. adding descriptors to the ring in which the network device can receive into). Similar the device can also return the buffers to the driver and hand back the ownership (remove a descriptor that was processed by the device).

To do some sanity checks on the transfers of ownership, we need to keep track of the memory that can be used for such a transfer. So in general before using a device queue, we register a large chunk of memory to it and then carve out buffers of this chunk of which we want to transfer ownership.

These operations we just described are represented by the following function calls:

```

1 errval_t devq_register(struct devq *q, struct capref cap,
2                         regionid_t* region_id);
3
4 errval_t devq_deregister(struct devq *q, regionid_t region_id,
5                           struct capref* cap);
6
7 errval_t devq_enqueue(struct devq *q, regionid_t region_id,
8                       genoffset_t offset, genoffset_t lenght,
9                       genoffset_t valid_data, genoffset_t valid_lenght,
10                      uint64_t misc_flags);

```

<sup>1</sup><http://www.barrelyfish.org/publications/TN-026-Device-Queues.pdf>

```

11 errval_t devq_dequeue(struct devq *q, regionid_t* region_id,
12                      genoffset_t* offset, genoffset_t* length,
13                      genoffset_t* valid_data, genoffset_t* valid_length,
14                      uint64_t* misc_flags);
15

```

The receive and send queue are represented by a struct `enet_queue` of which the first element is a `devq` struct. This allows for C style inheritance since if we cast the struct `enet_queue` to a struct `devq` the struct fields are accessible as usual. In this manner, you can use the `enet_queue` structs with the device queue interface.

Before enqueueing a buffer, you will have to register a region to the device queue. The function `devq_register()` will do this for you. The argument is a cap to some allocated memory and it returns you a region id. The returned region id can then be used to transfer ownership of part of that region. The size of the registered region should be at least in the order of what the descriptor queues can store. In the case of our ENET network device the queues have each 512 entries and each entry can point to a packet with the Ethernet MTU of 1536 bytes. Most network stacks round this number up to the next power of two (2048 bytes).

Keep in mind that you will have to keep track on your own which buffers are currently owned by the device and which buffers are still owned by the process. If you enqueue a buffer twice before dequeuing, this can lead to undefined behavior. It might make sense to write some code/library to do this for you. For debugging purposes you can also stack our debug interface on top of the queues (`include/devif/backends/debug.h`) which will return an error if the buffer is not owned by the process trying to enqueue/dequeue the buffer. The `debug_create` function takes another device queue (in our case this would be the `enet_queue`) and simply "bends" some function pointers so that first the debug queue code will be executed and then following it the send/receive queue code. A valid design for the whole network stack could be based on the stacking of all the different network layers on top of each other similar to the debug queue.

After registering, you can enqueue/dequeue buffers from the queue. For the receive queue, enqueueing a buffer means handing empty buffers to the hardware queue. The receive hardware queue will then fill these buffers with the data of received packets. When there is a packet in the descriptor queue, dequeue will not return an error and the argument pointers contain the information of a valid buffer. Similar for sending hardware queue, enqueueing means adding a buffer with a valid packet (including Ethernet, IP etc. headers) to the descriptor ring. When the buffer is sent, you will be able to dequeue the sent buffer from the queue and you can reuse this buffer.

Note that the arguments to enqueue has more than simply the start address and length of the buffer. A buffer in the device queue interface is described by `offset`, `length`, `valid_data`, and `valid_length`. The `offset` is relative to the start of the region which means you need to know the virtual address of the beginning of the region. The buffer is then defined by `start + offset`. Additionally, to facilitate stripping and adding headers, there are the two arguments `valid_data`

and valid\_data. The value of valid\_data represents the offset into the buffer (not the region!) at which valid data is present. By adding or subtracting from valid\_data you can add/remove headers easily and then hand off the buffer to the next layer. Similarly, valid\_length represents who much data in the buffer is valid since the buffers that you normally use will be larger than the packet that your receive/send.



*Project  
Instructions*

## 11.5 Implementing ARP

To send a packet to another machine, you normally only need to known the IP address. If you build the network stack yourself, you also need to known the MAC address of the target machine. To figure out the correspondence between the IP address and the MAC address, the ARP (Address Resolution Protocol [87]) is used.

We provide you with the header formats of all the different packet types you need to build/decode. The headers are located in `include/netutil/`

Packet of type ARP can be quickly identified by the type field of the Ethernet header. For an ARP packet the type field is 0x0806 (similarly for IP it is 0800). Keep in mind that Packets are always encoded in the network byte order, which is big-endian. Your system uses little-endian, so you have to convert between host and network byte order. To do so, you can use the `ntohs` family of functions. In the function `ntohs`, `n` for network, `h` is for host, and `s` for short. Hence you should use this to convert a 16bit value from your network to to the host byte order. We provide an implementation of these functions in `netutil/htons.h` where also the reverse functions `htons` and functions for 32 bit values are implemented.

The ARP protocol has two types of messages that have the same header format: request and reply. A request has the opcode field set to 0x1 and the reply to 0x2.

Normally a network stack does not only reply to ARP requests designated to the IP of the machines network interface, but also cache any information from ARP replies in a cache. One of the collection libraries in the handout might be useful (`include/collections/`) to implement this cache. The cache should simply store the translation from MAC address to IP address.

Similarly, you can store the information from requests, but you also need to check if the destination ip of the header is the IP of your network interface. You can use a static IP (e.g. 10.0.2.1) for your network interface and match against that one. If the ARP request is designated for your stack, then you need to fill in the fields of the ARP and Ethernet headers to properly respond to the request. The headers of packets are nested and the outermost header is Ethernet. The Ethernet header should be straight forward to build. The provided packet formats also contain many defines that will help you figure how to compute the offset to where the ARP header should start (offset from the start of the Ethernet header) but also values for differ-

ent header fields. Wikipedia is also pretty helpful in that regard<sup>2</sup>. The essential part is that you set the source IP and source MAC to the right values. As mentioned before you can use a static IP as your source IP. The MAC address of your network device is available from the `enet_driver_state` struct (`st->mac` in the main function of the driver). The destination information is available to you from the received request header.

If you send out packets yourself (e.g. ICMP response), you will also need to send ARP requests to figure out the MAC address of the destination network device or query your cache. If the MAC address is not in your cache, you will have to broadcast an ARP request by setting the destination MAC to `0xFFFFFFFF` and the destination IP to the address you want to query for.

## 11.6 Reply to echo requests (ping)

The next step is to decode the IP/ICMP packet, and write back a correct response. ICMP is standardized in RFC 792 [90], although Wikipedia has a good explanation of how Ping packets work<sup>3</sup>.

If you encounter a bitmask however, you must also take into account the bit order. For instance, to set the DF flag which is the first bit in the offset field of the IP header, you will have to set the *first bit from the left*. This has the byte representation `0x40`, to make it end up at the first position, it has to be the higher byte of the short that you write into the offset field. Hence to set the DF flag you can use `ip->offset = htons(0x4000);`

The maximum size of an IP packet is 64KiB. When the underlying link layer does not support packets that big, *IP fragmentation* allows to split IP packets in smaller packets. You do *not* have to implement fragmentation, but you should detect and drop incoming fragmented packets.

The IP header contains a checksum. The checksum algorithm is defined in RFC 1071 [31]. We provide an implementation of the checksumming function in `netutil/checksum.h`.

As previously mentioned you can assign a static IP to your Toradex board. You have to set this as source address in your IP packets. You should not hard-code the host address when replying to echo request messages.

Keep in mind that you also need to fill in the headers for Ethernet. You should now be able to successfully ping your Toradex board from the host.

## 11.7 Debugging hints

Debugging network protocols is inherently hard: they are asynchronous, and bugs often appear at the interaction between code on two different machines rather than



Project  
Instructions



Technical  
Details

<sup>2</sup>[https://en.wikipedia.org/wiki/Address\\_Resolution\\_Protocol](https://en.wikipedia.org/wiki/Address_Resolution_Protocol)

<sup>3</sup>[https://en.wikipedia.org/wiki/Ping\\_\(networking\\_utility\)#ICMP\\_packet](https://en.wikipedia.org/wiki/Ping_(networking_utility)#ICMP_packet)

in a single program.

We strongly recommend the use of `tcpdump` or Wireshark [101] to see what's going on, at all stages of this project. Not only can it decode packets according to the indicated protocol, but it can be used to inspect packets which don't conform and help you figure out what you're doing wrong.

Writing test programs is always a good idea, but particularly helpful here. Some code to send out packets from your OS with particular properties to a Wireshark instance running on the other end will give you confidence that you're formatting your packets correctly.

Note that you'll need to lay out packet headers correctly in memory. To write structs that match the UDP/IP headers, remember `gcc`'s support for `__attribute__((__packed__))`. You can also write a Mackerel definition. See the files in `devices/` as an example.



*Project  
Instructions*

## 11.8 UDP echo server

The next step is to create a UDP echo server that listens on a port and replies to the sender by sending back the same packet. You can hardcode the port on which the server is listening. Use the IP/UDP header information source information to determine the destination address and port.

Use the following `netcat` command to test your echo server. It will open a UDP connection and send a packet once you press return. It also displays any answer that your server sends back.

```
$ nc -u 10.0.2.1 Port
```

You should be able to demonstrate with the command above that your implementation is functioning properly.



*Project  
Instructions*

## 11.9 Multiple clients

For this section, you will export a service to other client applications. The client should be able to send UDP packets and listen for UDP packets on a specific port. As a demo application move your echo server into a separate process. You should be able to demonstrate that you are indeed running the server in a separate domain by first pinging your machine without the echo server running. Then, after starting the echo server, it should also act as an echo server.

The final step is to allow multiple echo servers running on different ports at the same time.



## 11.10 Performance measurement

We are not expecting terrific network performance from the 100 Mbit network device of the SOC, but we do expect you to measure it, and comment on where the time is going. As with most network measurement, the basic quantities are:

**Bandwidth:** Can you saturate the network link with packets? If not, why not? Do packets get dropped under high load? etc.

**Latency:** How long does it take to get a single packet from a user-space program on your board to another one on the host?

Include figures, graphs, and other measurements in your chapter of the final report.



## 11.11 Remote UDP login

If one of your fellow team members is working on a shell (and hopefully one of them is), then you can try to implement a simple remote login over UDP. For minimal functionality, ignore the fact that UDP packets can be dropped and assume reliable delivery.

Your implementation will listen on a port for commands, and then execute them in the shell before returning the output.

However, if you are feeling more ambitious and have the time, you might want defer remote login in favor of the next optional challenge: TCP.



## 11.12 TCP

If you are feeling up to it, you can try to implement a minimal TCP. A full TCP implementation is really rather complex, and needs careful attention to the various timers used in the protocol, slow start, Nagelling, and all the various other enhancements to the basic protocol. However, it is not too difficult to get *something* working which talks to a correct TCP implementation running on a host: TCP is highly robust to incomplete implementations.

One of the best descriptions of TCP is in Steven's classic three-volume book on network protocols [109, 110]. To start with, don't worry too much about timing, and instead focus on getting a basic 3-way handshake working with your implementation as the client, connecting to a suitable server on your host (again, `netcat` is very useful for this). Once you can establish a connection, try sending packets in both directions.

After you've got this bit working, build the server-side: accepting connections on a port, and creating a new TCP control block for each new connection.

Finally, try to get connection teardown working. This is quite complex from a state-machine perspective, but certainly tractable. For this, however, you'll probably want

some kind of basic support for timeouts.

To show off your TCP, you can use it for remotely accessing the shell, or even implementing a simple web server. For even more points, however, measure its performance, and show how well it performs under high load.



### Commentary

## 11.13 How to do well

As with the other individual projects, integration with the other projects of your team members is a big win here. Being able to demonstrate shell commands (or, ideally, binaries loaded from a file system by the shell) for operations like ping or sending or echoing UDP packets, etc. makes for a great demo.

Measurements are also important. It's great to be able to show working protocol implementations, but it's equally valuable (including points!) to analyze the performance of your implementations. It doesn't matter so much if it's slow, what's more important is that you can explain why using measurement evidence.



### Project Instructions

## 11.14 Summary

You are expected to demonstrate the following functionality:

- Displaying that you handle the ARP protocol correctly by showing the entries in your cache and also on the other machine connected to your Toradex board (`arp -n` can help you here).
- Dump a small ping packet to standard output.
- Show that you can ping your system.
- Send and receive back a packet from the UDP echo server by using `netcat`.
- Show your echo server is running in a separate domain, by starting the process at runtime.
- Start two different UDP echo server processes on different ports.

Additional challenges:

- Remote command line access over UDP
- Demonstrate TCP connectivity

# Chapter 12

## Nameserver

This individual project is about building a name server, which is a key central component of any non-trivial distributed system and, increasingly, most operating systems as well.

You have already seen how different functionality in a microkernel or multikernel is implemented in server processes. Name servers address the problem of how a process that provides a service (the server, in other words) can be discovered and contacted by a process that wants to use that service (the client). The way a name server works is that other processes in the system tell the name server about how to bind to the services they provide, and also ask the name server about how to bind services they need. In this way the name server acts as a “clearing house”: matching offers of services to requests for them.

It should become clear during this project (if it’s not already clear) that there is a large design space for name servers. This project will leave you much freedom in how to design and implement a name service, and also give you a lot of room for additional bonus objectives. As always, try to finish the minimal requirements first before starting with other projects.

The work consists of:

- Creating a separate server process (the *nameserver*) which can respond to requests
- Providing *lookup* functionality: clients should be able to connect to the name server, submit name lookup requests, and receive back enough information to allow them to connect to another service.
- Providing *registration* functionality: servers should be able to connect to the name server and register their service IPC endpoint with the server under a particular name, so that clients can then query it.

- Create a mechanism for *bootstrapping* the name service: since a new client can't look up the name of the nameserver, there has to be a mechanism for any process to acquire a connection to the nameserver when it starts
- Demonstrating all other processes in the OS as a whole using the name-server rather than other mechanisms for obtaining IPC channels.



*Project  
Instructions*

## 12.1 Getting prepared

Conceptually, a name server can be as simple as a hash table from strings to endpoint identifiers (for example, capabilities). The subtlety is in the design of the naming scheme, and how the name server interacts with the rest of the system.

Name servers in modern *distributed* systems support many complex features like replication (for performance or fault tolerance), delegation (such as DNS' hierarchy), security (not all clients of the name server can see all services registered with it), etc. For your OS, these extra features are *not* required - a small OS simply doesn't need them.

However, even small microkernel or multikernel-based OSes really need some kind of name server to be practical. Since your fellow team members are implementing other functionality for their projects, it's quite likely that they will find early access to your name server's code useful for their own work.

There is no pre-existing code required for this milestone. All you should need is the code that you and the rest of your team have written up to this point. However, by the end you should also be able to integrate the code that your fellow team members are writing (see below).

Instead, the name server will be written "from scratch", but building on the communication primitives you have already seen. The challenges are in figuring out how to use these communication operations to implement the name server functionality, and how to organize the name server and its API.

This process may uncover deficiencies and limitations in the code you have already written, although if you have designed and implemented your previous milestones well enough, this should not be problem. If it is, you should be prepared to fix these issues, but also talk about what you did in the project report and why it was required.

We suggest you start with a very simple example server which can be contacted by other processes and serve requests, and then extend your code from there into a fully-fledged name server.

You will also want to write a small library that clients of the name server use to communicate with it, whether they are registering a service or looking one up. In the rest of this handout we'll assume the API for this library is defined in a header file include/nameserver.h.

## 12.2 Registration

The first functionality to provide in the name server is registration of new services.

```

1 typedef (void *)nameservice_chan_t;
2
3 ///< handler which is called when a message is received over the registered channel
4 typedef (void)(nameservice_receive_handler_t)(nameservice_chan_t chan,
5                            void *st, void *message,
6                            size_t bytes);
7
8 /**
9  * @brief registers our selves as 'name'
10 *
11 * @param name our name
12 * @param recv_handler the message handler for messages received over this service
13 * @param st state passed to the receive handler
14 *
15 * @return SYS_ERR_OK
16 */
17 errval_t nameservice_register(const char *name,
18                            nameservice_receive_handler_t recv_handler,
19                            void *st);
```



Project  
Instructions

The `register` function is called by a server (for example, a file system server) to announce that it's providing a service that clients can connect to. The first argument is the name that the service will be registered under. The remaining two arguments are a function pointer to a receive handler for messages which arrive for the registered service, and a state pointer which is passed to the handler function.

It is up to you, to decide how to integrate this in the messaging framework you have developed so far. Keep in mind, that there may be many different services running on your system, all of which having slightly different requirements and thus message types. One way to implement this is to have a specific message in the existing messaging layer you have written so far, which encapsulates the message for the registered services. You are free to decide how to do this, in the end the correct registered receive handler should be called, also when there are multiple services running in one domain. It is up to you if you want to implement this as separate UMP/LMP channels, or implement a multi-hop protocol through `init`.

It is also up to you to design the *name space* to be used. A bare minimum is simply flat strings (such as “`memserver`” or “`face_detect`”); but even here you need to specify what is a valid name and what is not. For example, what kinds of characters are allowed in names? What about whitespace?

More advanced name servers use a hierarchy, like DNS or POSIX file system names, with some character separating *components* of the name (for example, “`/fs/sdcard`”). This makes it easier to manage the name space since different areas of functionality can be given different subtrees of names.



*Extra  
Challenge*

### 12.3 Rich properties

Really sophisticated name servers support much more than simple strings for naming – they are closer to structured search engines. For example, it is quite common to use a set of key-value pairs instead of a single string, such as:

```
type=ethernet
mac=44:8a:5b:d3:b8:07
name=eth0
speed=1G
```

The lookup operation (see below) is then a search for records which satisfy some query, such as “An ethernet interface with a speed of 1G”. If simple unique names are enough, they are a subset of this functionality (such as searching for a record with field `name` equal to `eth0`).

Such sophistication might not be needed for your system, but you might want to see how easy it is to implement it in some way.

As well as registering services, you should allow a server to deregister them – remove any record of them from the name server, using the `deregister` call.

```
1 /**
2  * @brief deregisters the service 'name'
3  *
4  * @param the name to deregister
5  *
6  * @return error value
7 */
8 errval_t nameservice_deregister(const char *name);
```



*Extra  
Challenge*

### 12.4 Dead service removal

Of course, if a server process crashes, then no client can contact it, even though the name server might have a record for it. Ideally, the name server would detect that the service has gone away itself and remove the “dead” service record.

Also, as specified above, anyone can call `deregister` with any name. It would be better if this could only be done by the server which originally called `register` for that name, or by an administrator account.



*Project  
Instructions*

### 12.5 Lookup

Once you can register (and deregister) services with particular names, the next step is to allow clients of the services to look them up by name in the name server:

```

1 /**
2  * @brief lookup an endpoint and obtain an RPC channel to that
3  *
4  * @param name name to lookup
5  * @param chan pointer to the chan representation to send messages to the service
6  *
7  * @return SYS_ERR_OK on success, errval on failure
8  */
9 errval_t nameservice_lookup(const char *name,
10                           nameservice_chan_t *chan);

```

lookup should take two arguments. The first is a query string. At its simplest, this query is simply a well-known name to lookup (as in the examples in the previous section), and we suggest you implement this simple interface first, before allowing more sophisticated lookup strings.

The remaining argument is an output, and returns something that can be used to connect to the service (assuming the lookup succeeds). Naturally, if the name is not bound in the server, an appropriate error code should be returned.

The aos\_nameservice\_chan\_t is an opaque type that references a channel to the service behind the registered name. This is used to send an RPC message to the service and sending a response back using the following interface (Note this may just be a wrapper around the aos\_rpc\_\*( ) functions):

```

1
2 /**
3  * @brief sends a message back to the client who sent us a message
4  *
5  * @param chan opaque handle of the channel
6  * @param message pointer to the message
7  * @param bytes size of the message in bytes
8  * @param response the response message
9  * @param response_byts the size of the response
10 *
11 * @return error value
12 */
13 errval_t nameservice_rpc(nameservice_chan_t chan, void *message, size_t bytes,
14                          void **response, size_t *response_bytes);

```

The idea behind the working of this is the following:

1. server: does the service registration, with a given callback.
2. client: looks up the service name and obtains the handle.
3. client: uses the handle and calls `nameservice_rpc()` with the handle.
4. server: invokes the receive handler

5. server: uses the `nameservice_send_response` to send a response to the client.
6. client: the call to `nameservice_rpc()` returns

See `/usr/namservice/nameservicetest.c` for an example.



*Project  
Instructions*

## 12.6 Bootstrap

Now that you have implemented registration and lookup, you have the basic functionality required to use the name server. However, a remaining problem is that of how each process acquires an initial connection to the name server.

Consequently, the next step is to make sure that, after a small number of initial processes (including the name server itself) have started, *all* new processes start with a way to connect to the name server.

You have most likely already solved this problem before in the case of the memory allocator. It should be possible to go back and modify the rest of the OS so that processes can locate the memory server after they have started up by first contacting the name server.



*Project  
Instructions*

## 12.7 Enumeration

By this point, you have almost certainly written debugging code to dump the state of the name server to make sure that registrations, etc. have worked.

The next step is to allow clients to obtain a list of all the names bound in the name server, or (better still) all names matching some search expression:

```

1 /**
2  * @brief enumerates all entries that match an query (prefix match)
3  *
4  * @param query    the query
5  * @param num      number of entries in the result array
6  * @param result   an array of entries
7  */
8 errval_t nameservice_enumerate(char *query, size_t *num, char **result );

```

Note that implementing this will require you to return an unpredictable (and potentially large) number of strings from the name server, which should probably be done using multiple messages.



*Project  
Instructions*

## 12.8 Getting the rest of the system to use the name-server

You have already implemented the core of a small operating system in the course of the project, and your project team partners will also be working on other components (such as a filesystem, shell, or network stack).

You should, as much as possible, get as much of the complete OS to use the name server as you can. This means that almost any service which can be contacted using inter-process communication (Local Remote Procedure Call on the same core, or User-level Remote Procedure Call between cores) should, by the time you are all finished, use the name server to register the services it offers, and look up the services it needs to use.

This isn't possible in all cases (for example, because the name server is itself a service that clients have to find), and you should make it clear in the report on your design where these limits are, if they could be overcome, and whether trying to do so would be a good idea.

However, ideally all new processes after the initial set should *only* need to start by contacting the name server. After that, all other services they should need can be obtained by looking up well-known names (or queries).

You should identify the minimal set of processes which, when the OS boots, need to have more than just a way to connect to the name server, and this set (and your reasoning!) should be documented in your report.

## 12.9 How to do well

There are two principal components to this project, and you will probably need to spend roughly equal amounts of time on both of them.

The first is designing and correctly implementing the protocols used to offer a service, look up a service, bind to it, and (optionally) remove an offer. We strongly suggest you work out the details of how all this works in advance, and write them down (you'll need them for the report anyhow).

The second is the user interface, if you like, to the nameserver: the query interface and how services are represented to clients. Here, it's about tasteful design. Don't be afraid to borrow concepts from other systems, and build a command-line interface to the name server both to help in debugging but also to demonstrate how it works.



*Commentary*

## 12.10 Interface to implement

See `include/aos/nameservice.h` and fill in `lib/aos/nameservice.c`.



*Project  
Instructions*



Project  
Instructions

## 12.11 Shell Integration to Implement

You should support the querying of the name service from the shell. Implement the following shell commands:

- `nslist` lists all the entries in the name server
- `nslookup <name>` looks up the name in the name server and displays the PID of the process running the service.



Project  
Instructions

## 12.12 Summary

You are expected to demonstrate the following functionality:

- An implementation of a name server supporting the four operations `nameservice_register`, `nameservice_deregister`, `nameservice_lookup`, and `nameservice_enumerate`, and the functions to send a reply message, and the above.
- The name server is started when the machine boots, and runs until the machine is shut down.
- All OS services register with the name server when they start up
- As many OS processes as possible use the name server to find out how to contact any other services that they need
- allow querying the name server from the shell.

You can demonstrate this in part by running a program (ideally, interactively from a shell that one of your fellow team members has written) that can list the services known in the name server.

Extra challenges:

- Show support a rich set of properties that can be associated with a service offer, and a corresponding query language that can match against those properties
- Demonstrate that the name server can handle removal of service offers, both via a process withdrawing an offer, or by detecting that the process is not responding or has crashed.

## Chapter 13

# Capabilities revisited

In previous milestones, you may have noticed that certain capability operations don't work, and always return an error, for example when you try to revoke a capability. This is because, those operations may require agreement and/or cannot be executed as a single step (e.g. delete a CNODE). In this project , the goal is to implement the missing pieces which are necessary for the capability system to work as a single entity across two cores of your system.

This individual project is less immediately “demoable” than others, but in some ways requires you to get to grips with deeper concepts. It gets to the heart of the character of a multikernel based OS like Barreelfish being somewhere between a shared-memory system and a classical distributed system, without being exactly like either of them.

The project is structured in such a way that there are logical steps that build on each other, and will have a lot of opportunities where you can implement functionality that is not necessary for the project work to be given a passing grade but which will certainly improve your grade assuming that all the core functionality outlined in the project goals is working correctly.

The work consists of:

- Implement transferring capabilities between the init processes on both cores.
- Forward libaos capability operations that need synchronization to init on the same core.
- Implement cap\_revoke for a single core.
- Implement cap\_delete for capabilities for which copies exist on both cores.
- Implement cap\_revoke for capabilities for which copies exist on both cores.
- Implement cap\_retype for capabilities for which copies exist on both cores.



Commentary

## 13.1 Background: Distributed capabilities

The CPU driver you have been using for the duration of the course already has a lot of the lowest level of functionality that is needed for Barrelyfish's distributed capability design. There are a fair amount of invocations that deal with different steps for these distributed capability operations. You will find convenience wrappers for those invocations in `usr/init/distops/invocations.{c,h}`.

The design utilized in Barrelyfish, which is what these invocations were designed for, has the following properties:

1. Each capability has an owner, which is one core in the system.
2. Each capability in each set of capabilities that refer to the same object ("copies") has the same owner.
3. The owning core for a capability  $c$  must always have at least one copy of  $c$ .

If you wish, you can use these three properties as guidelines for your own design.

Given those three properties, things you need to think about are how to manage ownership in the case where you delete the last copy on the owning core, and whether it might make sense to voluntarily move ownership between cores. Terminology you will see in the provided code is that capability copies on the owning core are sometimes called "local", which capability copies on non-owning cores are sometimes called "foreign".

If you want to invent a different design yourself, you should be aware that you might have to modify the CPU driver for your system to be able to implement all the required operations. Additionally, a lot of the information presented below will not necessarily apply for your design, and you will have to make sure that "normal" capability invocations, cf. `lib-aos/capabilities.c` and `kernel/syscall.c`, will correctly signal user space whenever an operation needs synchronization such as when a direct invocation returns `SYS_ERR_RETRY_THROUGH_MONITOR`.



Project  
Instructions

## 13.2 Getting prepared

For this milestone, you will need to obtain some additional code. Check whether you have `/usr/init/distops` in your tree, if not pull the handout repository to obtain the additional files, which will add a number of new invocations and other supporting code to `init`. Don't forget to add them to the Hakefile of `init`.



Project  
Instructions

## 13.3 Transferring capabilities between the init processes on both cores

A good place to start your implementation of a distributed capability system is to actually be able to correctly send capabilities from `init` on one core to `init` on the

other core. This involves making sure that each capability's owner information is set to the core which will be responsible for synchronizing complicated operations on that capability. In the provided code there are a number of functions that will be useful to manage ownership and remote relations of capabilities:

#### Ownership and remote relation management methods

```

1 errval_t monitor_get_cap_owner(struct capref croot, capaddr_t cptr,
2                               int level, coreid_t *ret_owner);
3 errval_t monitor_set_cap_owner(struct capref croot, capaddr_t cptr,
4                               int level, coreid_t owner);
5 errval_t monitor_remote_relations(struct capref cap,
6                                   uint8_t relations, uint8_t mask,
7                                   uint8_t *ret_relations)

```

As you can see, these functions have “monitor” in their function names, hinting at the fact that in Barreelfish they’re used in the *monitor* which is the domain running on each core in the system that executes privileged operations with unbounded execution time, so most of the CPU driver operations can be completed in bounded time, allowing the CPU driver code to run with interrupts disabled.

What you actually send to the other core is the actual struct capability of the capability that you are transferring. In order to create a correct capability on the receiving core you will use `monitor_create_cap()`, which is similar to the function you are using to create a RAM capability for the memory server on core 1.

Given the design properties mentioned earlier, you can see that you will need to properly set the owner of each capability to either core 0 or core 1. The remote relations for each capability indicate whether a capability has copies, ancestors, and descendants on the other core.

Helper functionality and bit definitions for the remote relations property of the capabilities is provided to you in the file `include/barreelfish_kpi/distcaps.h`.

Assuming you implement this capability transfer correctly, you will get the error `SYS_ERR_RETRY_THROUGH_MONTIOR`, not only for revokes but also for `cap_retype` and `cap_delete` if those functions are called on a capability for which copies exist on both cores.



*Extra  
Challenge*

## 13.4 Ownership transfer

Once you can transfer capabilities between the two cores you may want to implement a variant of the capability transfer which makes the receiving core the new owner of the transferred capability.



*Project  
Instructions*

## 13.5 Forward libaos capability operations that need synchronization to init on the same core

To handle these operations that return RETRY\_THROUGH\_MONITOR, you will have to implement a set of RPCs to init which will act as the synchronization point on each core for capability operations that need synchronization.

This can be done quite easily by providing init access to your domain's cspace. This can be accomplished by an RPC call that sends the domain's root cnode (cap\_root) to init and further provides capability address of the capability on which the operation was originally invoked as an argument.

In this step it is sufficient for init to print out some information about the capability which needs to be retyped/deleted/revoke. From init, you can easily refer to capabilities that exist in other cspaces by creating a domcapref from the information received through the RPC (cf. `usr/init/distops/domcapref.h`). Init also has the means to inspect capabilities by using the privileged invocations `monitor_cap_identify()` and `monitor_domains_cap_identify()`, which you can use to acquire a copy of the actual struct `capability` from the CPU driver. To produce a human-readable format from that information there is a function `debug_print_cap()` in libaos.



*Project  
Instructions*

## 13.6 Deleting CNodes and Dispatchers

For now, we are taking a step back and are looking at the case where we delete the last copy of a capability that itself contains references to other capabilities. This capability is most likely a CNode, but the Dispatcher control block capability also contains a copy of the dispatcher's root CNode.

Assuming that the capability is a CNode, and the copy we are deleting is the last copy, we need to make sure that all the capabilities that are contained in that CNode are deleted as well.

As we cannot reliably predict the execution time for such an operation, we split this operation into multiple steps. The first step is iterating through the CNode and marking all the capabilities it contains for deletion. This step is actually done by the CPU driver when init calls `monitor_delete_last()`. After that first step there is one step for each capability that got marked for deletion. These steps can be executed by repeatedly calling `monitor_delete_step` and `monitor_clear_step`. This will process capabilities that are marked for "delete" and "clear" respectively. Clearing a

capability is the step where the CPU driver will create a new capability from scratch if there is no capability referring to the underlying object left in the system.

### 13.6.1 Delete Stepping Framework

For your project, we provide a set of functionality that wraps the delete and clear steps in a easy-to-use event-based framework, which you can find in the file `deletestep.c` in `usr/init/distops`. In order to use the framework you need to initialize it by calling `delete_steps_init()` with a waitset on which you are doing `event_dispatch()` as the argument (e.g. the waitset you are using to process events on LMP channels).

#### Delete stepping methods

```

1 void delete_steps_init(struct waitset *ws);
2 void delete_steps_trigger(void);
3 void delete_steps_pause(void);
4 void delete_steps_resume(void);
5 void delete_queue_wait(struct delete_queue_node *qn,
                      struct event_closure cont);
6

```

If the framework is initialized, processing delete and clear steps is a matter of setting up a callback which will be called when the delete and clear queues are empty by using `delete_queue_wait()`, and making sure that queue processing is paused and resumed at appropriate points using `delete_steps_pause()` and `delete_steps_resume()` (processing delete steps while we are inserting new elements might be a bad idea).

## 13.7 Revoking capabilities

When revoking a capability there is always the possibility of suddenly having more work of our hands. Consider the case where we revoke a RAM capability that was retyped to a CNode. Now all the issues discussed in the previous section apply to that revoke, as we need to make sure that before we finally delete a CNode all its contents are properly deleted.

For revoke we again split the work in multiple steps. The first step is marking all copies and descendants for deletion with `monitor_revoke_mark_target`. After that we need to process the delete and clear queues the same way we did for delete.



*Project  
Instructions*



## 13.8 Deleting capabilities that exist on both cores

As you can imagine, deleting a capability for which copies exist on both cores needs synchronization. One reason we need synchronization is that what we discussed in section 13.6. A further complication is rule #3 in our example design “The owning core of a capability  $c$  must always have at least one local copy of  $c$ ”. This means that deleting the last copy of a capability on its owning core in the presence of copies on the other core needs to transfer ownership to the other core (or delete all copies on the other core, if ownership for the capability’s type is not transferable).

Exactly how you do this synchronization is up to you. Barrelyfish’s implementation, which needs to worry about systems with more than two cores does all operations that require synchronization as a two-phase commit (2PC).

2PC matches nicely with the mark & sweep scheme discussed for deleting and revoking capabilities in the previous sections. The mark phase is the first phase of the two-phase commit and we trigger the sweeping when we send or receive the commit of the 2PC protocol.

This protocol needs to happen between the init processes and to implement the protocol you can send messages and/or RPCs over the UMP channel between the two init processes.



## 13.9 Revoking capabilities that exist on both cores

As you can imagine, doing a capability revoke – which is essentially a delete of all copies and descendants of the target capability – in a system where copies and descendants of the target can exist on multiple cores needs to be synchronized for the same reasons an individual delete needs to be synchronized.

We can also use the same 2PC synchronization where we mark first and then sweep when we get the commit.



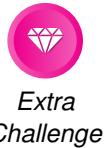
## 13.10 Retype capabilities that exist on both cores

When we try to retype a capability for which copies exist on both cores it is necessary to check that the requested retype doesn’t violate the guarantees the capability system makes about types and never allowing capabilities to refer to overlapping regions. The only case where we allow overlap is for parent capabilities, which must completely contain the regions referred to by their children.

In order to check this for this overlap, we need to forward the retype request to the other core which then needs to check that it does not hold any capability that would produce a retype conflict. To do this check you can use `monitor_is_retypeable()`, which will execute all the checks that are done during a `cap_retype()` invocation, but will not actually produce the retype result.

## 13.11 Use standard capability operations from init

As you may have noticed, init itself does not work when using a libaos capability function that causes an RPC. A general solution to this problem – using the same library code in the domain handling the RPC as in all other domains – is to provide a special RPC backend which just locally calls the right handlers. Implement such a “local” or “loopback” RPC backend, so init can use standard `cap_revoke` and `delete`s and retypes for capabilities with copies on both cores.



## 13.12 Use “real” capability transfer when initializing init on core 1

Now that you have a completely working distributed capability system, you can use your cross-core cap transfer to send the initial RAM cap and the bootinfo capabilities to init on core 1.



## 13.13 Send capabilities between any two domains

Now that you have a completely functional distributed capability system, you can also look at sending capabilities from a domain on one core to a domain on the other core. This must involve init on both cores as you cannot transfer capabilities between cores without having a trusted domain to create the capability on the destination core. It is perfectly acceptable that any message between domains on different cores which contains a capability is routed via the init domains, even if you have direct UMP channels between the domains. However, this decision should be made transparent to the client code.



## 13.14 How to do well

Tests, tests, tests!

This project is dealing with quite subtle interactions, and there is a lot of scope for tricky bugs to emerge. Consider using some or all of the following techniques to get this right:

- Write down the set of *invariants* over the distributed capability database that must be enforced at all times.
- Write code to check these invariants.
- Be able to visualize all or part of the capability system to help figure out what has gone wrong if the invariants are violated.
- Simulate the capability system inside a suitable harness on Linux to help with debugging, before transferring it to Barreelfish.



- Consider writing a formal specification of the operations and their semantics, in a language like TLA+ [69]



*Project  
Instructions*

## 13.15 Summary

You are expected to demonstrate the following functionality during the lab-session:

- You can send capabilities between the two init processes
- You can demonstrate init receiving and processing capability operations
- You can correctly handle cap\_delete for any capability
- You can correctly handle cap\_revoke for any capability
- You can correctly handle cap\_retype for any capability

Additional challenges:

- Optionally transferring ownership of the capability between cores.
- Allow the init domain to use standard cap\_retype, cap\_delete, and cap\_revoke on capabilities which require synchronization.
- Change your system to use your capability transfer functionality to provide a large RAM region and the bootinfo structures to init on core 1, instead of whatever scheme you originally implemented in Chapter 7.
- Send a capability from an arbitrary domain on core 0 to an arbitrary domain on core 1.

## **Part IV**

# **Group work: The Report**



# Chapter 14

## The Report

The final deliverable in the course is the report. Although it is the last, it is the most important for several reasons: it is, in part, a record of everything else that you have done so far in the course, and it carries a disproportionately high percentage of the total marks for the course.

### 14.1 Structure of the report

Like the course itself, the report typically has several different elements:

1. A detailed description of the core of the OS you built as a group.
2. A section containing descriptions of the individual projects you undertook.
3. An evaluation of the performance of the OS you have built.
4. A user guide for the OS, explaining how to run its functionality and any particularly important programming interfaces



*Project  
Instructions*

A good report is quite large. Font sizes and margins vary, but if you end up with between 70 and 100 pages with reasonable formatting, you are probably on the right track.

Typically teams have a different chapter of the report for each milestone, and for each individual project. The user guide is typically in an appendix.

### 14.2 Group Milestones

Your report should include a significant write up of each of the group milestones which build the core of the OS. We'll assume that you collectively wrote this part of the report as a group, and grade you accordingly.



*Project  
Instructions*

This part of the report should describe, for each group milestone, what you eventually implemented, and why.

It should also include anything else which shows off your understanding the problem being solved, and the process you went through to solve it.

Do *not* use documentation generated from the code. Feel free to include this, but at the end of the report as an appendix.



#### Commentary

### 14.3 Documenting a design

Documenting a design (which is what most of the report is) is not the same as documenting an API or writing a user guide. It should discuss things like:

- Algorithms
- Invariants
- Performance goals
- How the system is decomposed into components at different levels
- How are threads allocated to tasks
- How do the various parts interact.

Try to keep code snippets to a minimum. It's OK to put code in where what it's doing is low-level and best described this way (e.g. an interrupt handler), but for most of the OS it's not the best way to describe things.

It is not easy to do a good job of documenting the detailed design of a complex system. You will need to break down the complexity without losing essential details. Various techniques have been devised for managing this complexity over the years.

The most natural one to most computer scientists is modularity: talk about the different components of the system, and their interfaces. This is good as far as it goes (and you should do this!), but has drawbacks:

- Focus on interfaces and modules has a tendency to emphasize API calls and not other important aspects of the design such as invariants and other global properties of the state.
- It is hard to convey the large-scale structure of a system using only modules, particularly if the dependency graph of modules is dense (as is often the case with OS kernels).

#### 14.3.1 Describing the system: layers

You're probably familiar with layered decomposition of systems from networking, where it's pretty much orthodoxy.

It's possible to view an OS, or parts of an OS, as a set of layers where, as in networking, each layer implements some functionality, exports this functionality to the layer above, and uses the functionality provided by the layer below.

Moreover, how this happens varies according to the kind of OS architecture adopted. A layered decomposition of a monolithic kernel like Unix, for example, would look very different from that of an exokernel-like system.

Unfortunately, many systems (in particular, distributed systems) don't perfectly decompose into layers, nor do layered decompositions capture everything important about the system. This means that layering might be a useful way to describe part of a microkernel or multikernel-based system, but it is hard to capture the whole system usefully this way.

### 14.3.2 Describing the system: viewpoints

A different way of describing a complex computer system was used by an old standard for distributed computing, the ISO/ITU "Reference Model for Open Distributed Processing", or RM-ODP [60]. RM-ODP is broken into 5 *viewpoints*, each of which focuses on a single facet of the whole system. In no particular order:

**Enterprise:** The Enterprise viewpoint is all about what the system is trying to achieve, and for whom.

For example, an Enterprise viewpoint for an OS would talk about its different relationships with the users of the OS, developers writing code for it, administrators or operators of systems running it, etc. and the functionality and guarantees it offers to them.

**Information:** The Information viewpoint is the complete high-level *state* of the system and how it can change, regardless of where the state is or what kind of communication is involved. The Information viewpoint is quite "declarative": it doesn't talk about functions, rather schemas and invariants.

For example, for an OS the Information viewpoint would talk about the set of applications running and their runtime guarantees, the set of network connections, mounted file systems, process hierarchy, etc.

**Computational:** The Computation viewpoint is about what *invocations* exist between different parts of the system.

In an OS, the semantics of the programmatic interfaces between system components and between applications and those components are part of the computational viewpoint. However, where the state sits in the system (how it is distributed) and what that state that is are not part of the Computational viewpoint (they are in the Information and Engineering viewpoints, respectively).

**Engineering:** The Engineering viewpoint is how all this fits together as a complete system: where is the data? How is it replicated? Which units can fail? etc.

For Barrelyfish, for example, the complete set of capabilities in the system and their relationships is part of the Computational viewpoint, but the distributed nature of the capability database (across the two cores in your system), and how they are kept consistent, is part of the Engineering viewpoint.

**Technology:** The Technology viewpoint is the details of how you actually build the system.

For example, the Technology viewpoint of an OS would talk about the use of C and assembly, optimized fast paths, memory layout, etc.

This is, of course, complete overkill for documenting a system the size of the one you've been building in this course. However, it's really useful to bear in mind this separation in order to make the documentation (and your own thoughts!) clear.

Another way of looking at this is that most systems documentation gets things wrong by confusing and conflating the Computational, Engineering, and Technology viewpoints into one, hard-to-understand API, and completely ignoring the Information and Enterprise viewpoints.



### Commentary

## 14.4 Documenting the process

A secondary purpose of the report is to document the process you went through in building the system.

For example, you have probably been performing benchmarking and/or profiling during the development of the system. If these results are interesting, include them in the description (unless they are really part of the performance evaluation, in which case they belong in the evaluation section).

Also, talk about alternative design choices: what options did you consider and reject? What went wrong when you tried some design element, and how did you figure this out? With the benefit of hindsight, what might have been done better?

Finally, include and call out performance optimizations you make. If you do, include the argument that they are correct - that is, they don't change the behavior of the OS except for making it faster or less memory hungry, etc. Say what you're optimizing for, and why.



### Project Instructions

## 14.5 Individual project write-ups

The report must also include a separate part (such as a chapter) on each of the individual projects built on the OS core. Each of these should be written by whoever wrote the code for that particular individual project.

The same requirements as for the core OS milestones apply here: the description should focus on the design choices.

However, since the individual projects are designed to fit together, you should also detail the dependencies your project has on the core OS *and* the other individual projects which make up the whole system, and vice-versa.

## 14.6 Performance evaluation

Computer systems care about performance, and operating systems are no exception. When making design choices, the factors to take into account should not be purely aesthetic: they should examine the trade-off between elegance, simplicity, reliability, and ease of use on the one hand, with efficiency and performance on the other.

For this reason, you've probably been measuring the performance of parts of the system you've been building all along, and you should also benchmark the complete system. All of these results should be in the report (note that there is no page limit).

Performance evaluation is a huge topic, and indeed there is an entire course at ETH Zurich (the Advanced Systems Lab) which is mostly devoted to it. You may have taken this course, or you might have not (yet), but here are some basics to bear in mind.

By no means the last word on performance analysis (especially these days), but a good place to start is Jain's book [61].



Project  
Instructions

## 14.7 Benchmarks

We typically refer to an experimental measurement of some aspect of system performance (throughput, latency, memory footprint, power consumption, etc.) as a *benchmark*, and distinguish between two types:

**Microbenchmarks** are fine-grained measurements of a small part of the system, for example, the cost of a single inter-process communication call, a thread context switch, or upcalling a dispatcher. Microbenchmarks are vital because they help to understand why the system as a whole is fast or slow, and what parts are responsible for the observed performance.

Microbenchmarks, on the other hand, convey no useful information about the performance of the system as a whole as seen by a user of an application.

**Macrobenchmarks**, in contrast, measure the complete performance of a system at some particular task. Good macrobenchmarks enable the performance of different systems (or different variations on one system) to be compared in the context of different applications.

The relationship between micro- and macrobenchmarks is subtle. For example, the performance of the Unix `fork()` system call is crucial to the performance of some



Technical  
Details

applications, but *not* a typical web server, which forks its worker processes once at the start of execution.

When presenting the results of a benchmark, make sure to state for each experiment the following four things:

- What this particular experiment is intended to demonstrate or measure. If you want to be really scientific, state the hypothesis that the experiment is designed to test.
- State what you did: the experimental procedure you followed, and what quantities you measured. This should be enough for someone else to reproduce the experiment.
- Describe the results: refer to graphs or tables, and point out the most interesting or significant features.
- Finally, draw your conclusions, which at least should relate directly to the purpose of the experiment. Additionally, if something really surprising happened, say so.



Technical  
Details

## 14.8 Performance methodology

Even for extremely low-level microbenchmarks, the value of each run of the benchmark is likely to yield a different value. For this reason, benchmarks are always run repeatedly. Since the first few runs are often in a different environment (for example, the cache might be empty), they are often discarded. In other cases, depending on the benchmark, each run is done on a “cold” system. It depends on what one is trying to measure.

You should perform enough runs of an experiment for the reader to be confident enough of the conclusions you draw from the measurements. If the result is highly deterministic (e.g. counting lines of code, or the number of machine cycles for an interrupt handler with a cold cache), you probably don’t need many runs to show that there’s really no variation, but make sure this is really the case.

However, if there is variation in your measurements, you need to be more clever. At a bare minimum, collect enough results from multiple runs to show that the variation is small. Calculate this variation.

To be fully rigorous, you can perform enough experimental runs until you can show statistically that the confidence with which you are claiming the experiment proves or disproves your hypothesis is above some threshold (such as 95% or 99% confidence).



## 14.9 Presenting performance results

If you've gathered a particular performance measurement (e.g. number of machine cycles for a null inter-process remote procedure call), and you're confident of the average value, you may be tempted to just give this average. Don't do this.

Is the "mean" value the appropriate one? What about the median? Or 99th-percentile? What would a consumer of these benchmark results really want to know? What is the tail of your measurement?

At a bare minimum, give standard deviations, and/or error bars in your plots showing confidence intervals and, ideally, maximum and minimum values.

Better still, where it makes sense, show a cumulative distribution function *and* give useful percentile values.

## 14.10 How fast does it need to be?

All this talk about performance measurement is probably giving you the impression that performance matters for this project. We are looking for a system that isn't pathologically slow, but this is not the full story.

Your OS is not necessarily going to be fast, nor do we expect it to be. What matters is that you understand how it was built, why it was built that way, and why it performs the way it does.

We will not penalize you if the design is basically sound, but the implementation is not as optimized as it might be (given, say, another 5 weeks) as long as you can explain in the report why it is the way it is. If you think it's slower than it should be, say so – this will only improve your grade.





Commentary

## 14.11 How to do well

It is hopefully clear now from this chapter what a really great report should contain. To summarize:

- Explain the design of the system, and don't just API calls (except in an appendix).
- Explain the process of building it, rather than presenting the single finished artifact. Say why you did things, and why you didn't do other things. If you made a mistake and had to go back and correct it in the design, talk about it.
- Include rigorous performance measurements, which you then explain.
- Show us that you've taken on the ideas in the course. Feel free to speculate about what might be good extensions or alternatives to what you eventually did.
- Make the report interesting – tell a story.



Project  
Instructions

## 14.12 Final milestone summary

Milestone tasks:

Your report is expected to contain the following:

- A section on each milestone (written collectively by the group), talking about the high-level design, choices you made, alternative designs, and any performance measurements or other experimental results you used to investigate trade-offs.
- A section on each of the individual sub-projects, written by whoever performed the work, detailing the same issues as the group milestones.
- A summary of the overall system design that resulted.

The following items are optional, and if included, they should be in an appendix:

- Function-level API documentation for the key interfaces in the system, for example generated from the code using tools like Doxygen.
- User guides or similar documentation for components such as the shell, filing systems, etc.

## 14.13 A final word

Congratulations on making it this far!

Our goal has been to give you a better understanding not just of operating systems (and ones that don't look like Unix), but also how systems people think about designing and building complex systems with plenty of moving parts, asynchrony, security concerns, performance requirements, and the like.

There isn't a single "right way" to do this course (though there are plenty of wrong ways), and this is partly why the report is so important: we want to see how you've thought about the problem and what you think of your own solution.

We're also interested in your thoughts about the course: what you particularly enjoyed, what you didn't, how it might be improved, etc. Please feel free to include a more general "experiences" section as an appendix detailing this: we will read it avidly.

We hope you've enjoyed the course, and also found it useful.



Extra  
Challenge



## **Part V**

# **Handing in your Code**



## **Chapter 15**

# **Preparing Final Submission**

*make sure the code works with our test programs, provide some simple tests in the handouts that exercise the API, have an automated test???*

*the students should make sure that it compiles, disable debug outputs*



# Bibliography

- [1] E. Abrossimov, M. Rozier, and M. Shapiro. Generic Virtual Memory Management for Operating System Kernels. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 123–136. ACM, 1989.
- [2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, , and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of Summer Usenix*, July 1986.
- [3] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.
- [4] A. Agarwal, L. Bao, J. Brown, B. Edwards, M. Mattina, C.-C. Miao, C. Ramey, and D. Wentzlaff. Tile processor: Embedded multicore for networking and multimedia. In *Proceedings of Hot Chips 19*, 2007.
- [5] Alpha Architecture Committee. *Alpha Architecture Reference Manual*. Digital Press, 3rd edition, 1998. ISBN-13: 978-1555582029.
- [6] Thomas Anderson and Michael Dahlin. *Operating Systems: Principles and Practice*. Recursive Books, 2nd edition, 2014.
- [7] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, February 1992.
- [8] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Scheduling in k42. White paper, IBM Research, 2002. <http://www.cs.bu.edu/~jappavoo/Resources/Papers/10.1.1.8.5274.pdf>.
- [9] Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, Amos Waterland, Robert W. Wisniewski, Jimi

- Xenidis, Michael Stumm, and Livio Soares. Experience distributing objects in an smmp os. *ACM Trans. Comput. Syst.*, 25(3), August 2007.
- [10] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. *SIGPLAN Not.*, 26(4):96–107, April 1991.
- [11] ARM Limited. *ARM Architecture Reference Manual*, armv7-a and armv7-r edition, 2014. <https://developer.arm.com/products/architecture/a-profile/docs/ddi0406/latest/arm-architecture-reference-manual-armv7-a-and-armv7-r-edition>.
- [12] ARM Limited. *ARM Architecture Reference Manual*, armv8, for armv8-a architecture profile edition, 2017. <https://developer.arm.com/documentation/ddi0487/gb/>.
- [13] ARM Limited. *ARM Cortex-A35 Processor Technical Reference Manual*, March 2017. Revision: r0p2.
- [14] ARM Limited. *ARM Power State Coordination Interface*, psci 1.1. release edition, 2017. <https://developer.arm.com/documentation/den0022/latest/>.
- [15] ARM Limited. *ARmv8-A Address Translation*, version 1.1 edition, 2017. <https://developer.arm.com/docs/100940/0101/armv8-A-address-translation>.
- [16] ARM Limited. *Memory management (Learn the architecture guide)*, version 1.0 edition, 2019. <https://developer.arm.com/architectures/learn-the-architecture/memory-management>.
- [17] ARM Limited. *SMC Calling Convention*, issue number 1.3 edition, 2021. <https://developer.arm.com/documentation/den0028/latest/>.
- [18] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS’13, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.
- [19] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP ’03, pages 164–177, New York, NY, USA, 2003. ACM.
- [20] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork() in the road. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS-XVII)*, Bertinoro, Italy, May 2019.
- [21] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on*

- Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.
- [22] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. why isn't your os? In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, pages 12–12, Berkeley, CA, USA, 2009. USENIX Association.
  - [23] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.
  - [24] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, Berkeley, CA, USA, 2014. USENIX Association.
  - [25] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 267–283, New York, NY, USA, 1995. ACM.
  - [26] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1):37–55, February 1990.
  - [27] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(2):175–198, May 1991.
  - [28] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.
  - [29] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984.
  - [30] Steve Blightman. Auspex Architecture – FMP Past & Present. Internal document, Auspex Systems Inc., September 1996. [http://www.bitsavers.org/pdf/auspex/eng-doc/848\\_Auspex\\_Architecture\\_FMP\\_Sep96.pdf](http://www.bitsavers.org/pdf/auspex/eng-doc/848_Auspex_Architecture_FMP_Sep96.pdf).
  - [31] R.T. Braden, D.A. Borman, and C. Partridge. Computing the Internet checksum. RFC 1071 (Informational), September 1988. Updated by RFC 1141.

- [32] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 12–25, New York, NY, USA, 1995. ACM.
- [33] David D. Clark. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, pages 171–180, New York, NY, USA, 1985. ACM.
- [34] Austin T. Clements, M. Frans Kaashoek, Eddie Kohler, Robert T. Morris, and Nickolai Zeldovich. The scalable commutativity rule: Designing scalable software for multicore processors. *Commun. ACM*, 60(8):83–90, July 2017.
- [35] Helen Custer and David A. Solomon. *Inside Windows NT*. Microsoft Programming Series. Microsoft Press, 2nd edition, 1997. ISBN-13: 978-1572316775.
- [36] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, New York, NY, USA, 2013. ACM.
- [37] Denx Software Engineering. Das U-Boot – the Universal Boot Loader. <http://www.denx.de/wiki/U-Boot/>, January 2020.
- [38] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 122–136, New York, NY, USA, 1991. ACM.
- [39] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. Spacejmp: Programming with multiple virtual address spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 353–368, New York, NY, USA, 2016. ACM.
- [40] Dhammadika Elkaduwe, Philip Derrin, and Kevin Elphinstone. Kernel design for isolation and assurance of physical memory. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, IIIES '08, pages 35–40, New York, NY, USA, 2008. ACM.
- [41] Yasuhiro Endo, Margo Seltzer, James Gwertzman, Christopher Small, Keith A. Smith, and Diane Tang. VINO: The 1994 Fall Harvest. Technical Report TR-34-94, Center for Research in Computing Technology, Harvard University, December 1994.
- [42] D. R. Engler, S. K. Gupta, and M. F. Kaashoek. AVM: Application-level Virtual Memory. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, HOTOS '95, pages 72–. IEEE Computer Society, 1995.

- [43] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, HOTOS '95, pages 78–, Washington, DC, USA, 1995. IEEE Computer Society.
- [44] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.
- [45] Dawson Engler, M. Frans Kaashoek, and James O'Toole. The operating system kernel as a secure programmable machine. In *Proceedings of the 6th Workshop on ACM SIGOPS European Workshop: Matching Operating Systems to Application Needs*, EW 6, pages 62–67, New York, NY, USA, 1994. ACM.
- [46] Hadi Esmaeilzadeh, Emily Blehm, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.
- [47] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and execution models in the fluke kernel. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 101–115, Berkeley, CA, USA, 1999. USENIX Association.
- [48] Florian Funke, Alfons Kemper, Tobias Mühlbauer, Thomas Neumann, and Viktor Leis. Hyper beyond software: Exploiting modern hardware for main-memory database systems. *Datenbank-Spektrum*, 14(3):173–181, Nov 2014.
- [49] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 87–100, Berkeley, CA, USA, 1999. USENIX Association.
- [50] John Giacomoni, Tipp Moseley, and Manish Vachharajani. Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 43–52, New York, NY, USA, 2008. ACM.
- [51] Ruth Goldenberg and Saro Saravanan. *OpenVMS AXP Internals and Data Structures, Version 1.5*. Digital Press, 1994. ISBN-13: 978-1555581206.
- [52] Steven M. Hand. Self-paging in the Nemesis Operating System. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 73–86, New Orleans, Louisiana, USA, 1999. USENIX Association.

- [53] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, October 1988.
- [54] Norman Hardy. Keykos architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, October 1985.
- [55] Tim Harris, Martin Abadi, Rebecca Isaacs, and Ross McIlroy. Ac: Composable asynchronous io for native languages. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA ’11, pages 903–920, New York, NY, USA, 2011. ACM.
- [56] Kieran Harty and David R. Cheriton. Application-controlled Physical Memory Using External Page-cache Management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 187–197, New York, NY, USA, 1992. ACM.
- [57] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, April 2008.
- [58] IBM Product Design and Development, General Systems Division. *IBM System/38 Technical Developments*. IBM Corporation, 1980 (1978). ISBN 0-933186-03-7. G580-0237-1.
- [59] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2A: Instruction Set Reference, A-M*, September 2014. Order Number: 253666-052US.
- [60] ITU-T (Telecommunication Standardization Sector of the ITU). *ITU-T Recommendation X.901: Information technology – Open Distributed processing – Reference Model: Overview*. X: Data Networks and Open System Communication – Open Distributed Processing. International Telecommunication Union (ITU), August 1997. International Standard 10746-1.
- [61] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1st edition, April 1991. ISBN-13: 978-0471503361.
- [62] Eliseu M. Chaves Jr., Prakash Das, Thomas J. LeBlanc, Brian D. Marsh, and Michael L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. *Concurrency - Practice and Experience*, 5(3):171–191, 1993.
- [63] Gerry Kane. *PA-RISC 2.0 Architecture*. Prentice Hall, December 1995.
- [64] Yousef A. Khalidi and Michael N. Nelson. The Spring Virtual Memory System. Technical Report SMLI TR-93-9, Sun Microsystems Laboratories Inc., February 1993.
- [65] Steve R. Kleiman. Vnodes: An architecture for multiple file system types in sun unix. In *Proceedings of the USENIX Summer Conference*, pages 238–247, June 1986.

- [66] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammadika Elkaduve, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.
- [67] Orran Krieger, Marc Auslander, Bryan Rosengrub, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a complete operating system. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 133–145, New York, NY, USA, 2006. ACM.
- [68] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 7:1–7:14, Berkeley, CA, USA, 2007. USENIX Association.
- [69] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 1st edition, July 2002.
- [70] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, April 1979.
- [71] Timothy E. Leonard. *VAX Architecture Reference Manual*. Digital Press, 1st edition, March 1987.
- [72] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.
- [73] D. Levi and D. Harrington. Definitions of Managed Objects for Bridges with Traffic Classes, Multicast Filtering, and Virtual LAN Extensions. RFC 4363 (Proposed Standard), January 2006.
- [74] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.
- [75] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.
- [76] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM.
- [77] Jochen Liedtke, Volkmar Uhlig, Kevin Elphinstone, Trent Jaeger, and Yoonho Park. How to Schedule Unlimited Memory Pinning of Untrusted Processes or Provisional Ideas About Service-Neutrality. In *Proceedings of the The*

- Seventh Workshop on Hot Topics in Operating Systems*, HOTOS '99, pages 153–, Washington, DC, USA, 1999. IEEE Computer Society.
- [78] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. Tessellation: Space-time partitioning in a manycore client os. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 10–10, Berkeley, CA, USA, 2009. USENIX Association.
  - [79] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 110–121, New York, NY, USA, 1991. ACM.
  - [80] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.
  - [81] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An idl for hardware programming. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, Berkeley, CA, USA, 2000. USENIX Association.
  - [82] Dejan Milojicic, Steve Hoyle, Alan Messer, Albert Munoz, Lance Russell, Tom Wylegala, Vivekanand Vellanki, and Stephen Childs. Global Memory Management for a Multi Computer System. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, pages 12–12, Seattle, Washington, 2000. USENIX Association.
  - [83] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, May 1990.
  - [84] NXP Semiconductor. *i.MX 8DualXPlus/8QuadXPlus Applications Processor Reference Manual*, rev. d, mx8dqxprm edition, November 2018.
  - [85] Elliott I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, MA, USA, 1972.
  - [86] Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. For a microkernel, a big lock is fine. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, pages 3:1–3:7, New York, NY, USA, 2015. ACM.
  - [87] D. Plummer. Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. RFC 826 (Internet Standard), November 1982. Updated by RFCs 5227, 5494.
  - [88] Andrei Poenaru. Explicit os support for hardware threads. Master's thesis, Systems Group, ETH Zurich Department of Computer Science, March 2017. No. 161.

- [89] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library os from the top down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 291–304, New York, NY, USA, 2011. ACM.
- [90] J. Postel. Internet Control Message Protocol. RFC 792 (Internet Standard), September 1981. Updated by RFCs 950, 4884, 6633, 6918.
- [91] Barreelfish Project. Hake. Barreelfish Technical Note 003, Systems Group, ETH Zurich, April 2010.
- [92] R. Rashid, Jr. Tevanian, A., M. Young, D. Golub, R. Baron, D. Black, W.J. Bolosky, and J. Chew. Machine-independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *Computers, IEEE Transactions on*, 37(8):896–908, Aug 1988.
- [93] Barret Rhoden, Kevin Klues, David Zhu, and Eric Brewer. Improving per-node efficiency in the datacenter with new os abstractions. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC ’11, pages 25:1–25:8, New York, NY, USA, 2011. ACM.
- [94] Michael D. Schroeder and Michael Burrows. Performance of the firefly rpc. *ACM Trans. Comput. Syst.*, 8(1):1–17, February 1990.
- [95] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: A fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP ’99, pages 170–185, New York, NY, USA, 1999. ACM.
- [96] System Architecture Group, University of Karlsruhe. The L4Ka::Pistachio microkernel. <http://www.14ka.org/65.php>, June 2004. Retrieved August 2017.
- [97] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, 4th edition, 2014.
- [98] Michael B. Taylor. Is dark silicon useful?: Harnessing the four horsemen of the coming dark silicon apocalypse. In *Proceedings of the 49th Annual Design Automation Conference*, DAC ’12, pages 1131–1136, New York, NY, USA, 2012. ACM.
- [99] Charles P. Thacker. Beehive: A many-core computer for FPGAs. Unpublished Microsoft technical note, 2010.
- [100] Charles P. Thacker and Lawrence C. Stewart. Firefly: A multiprocessor workstation. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS II, pages 164–172, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

- [101] The Wireshark Team. Wireshark. <https://www.wireshark.org>, November 2017.
- [102] Toradex AG, Altsagenstrasse 5,6048 Horw, Switzerland. *Colibri iMX8X Datasheet*, 0.94 edition, December 2018.
- [103] Toradex AG, Altsagenstrasse 5,6048 Horw, Switzerland. *Aster v1.1 Preliminary Datasheet*, 1.5 edition, May 2019.
- [104] Unified Extensible Firmware Interface (UEFI) Forum, Inc. *Unified Extensible Firmware Interface (UEFI) Specification*, March 2019. Version 2.8.
- [105] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A note on distributed computing. In Jan Vitek and Christian F. Tschudin, editors, *Mobile Object Systems - Towards the Programmable Internet, Second International Workshop, MOS'96, Linz, Austria, July 8-9, 1996, Selected Presentations and Invited Papers*, volume 1222 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 1996.
- [106] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10*. RISC-V Foundation, May 2017.
- [107] David Wentzlaff, Charles Gruenwald, III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An operating system for multicore and clouds: Mechanisms and implementation. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10*, pages 3–14, New York, NY, USA, 2010. ACM.
- [108] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture, ISCA '14*, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.
- [109] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated*, volume 1: The Protocols. Addison-Wesley Professional Computing, 2nd edition, January 1995.
- [110] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated*, volume 2: The Implementation. Addison-Wesley Professional Computing, 2nd edition, January 1995.
- [111] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *Commun. ACM*, 17(6):337–345, June 1974.
- [112] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *Proceedings of the 2016 USENIX Conference*

- on Usenix Annual Technical Conference*, USENIX ATC '16, pages 309–322, Berkeley, CA, USA, 2016. USENIX Association.
- [113] Gerd Zellweger, Simon Gerber, Korniliос Kourtis, and Timothy Roscoe. Decoupling cores, kernels, and operating systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 17–31, Berkeley, CA, USA, 2014. USENIX Association.