Advanced Operating Systems

—

Final Report

Team 05

Tobias Buner Miro Haller Gyorgy Rethy
Marc Rettenbacher Patrick Ziegler

Spring Semester 2021

Contents

Table of Contents i

I Individual Early Milestones 1

1 Milestone 1: Memory Management and Capabilities 2
1.1 Physical Memory Management . . . 2
1.1.1 Allocation . . . 3
1.1.2 Freeing . . . 4
1.1.3 Slot- and Slab-allocator . . . 4
1.1.4 Minimal Block Size and Alignment . . . 5
1.1.5 Team001 Comparison: Alternative Memory Manager Design Evaluation . . . 6
1.2 Mapping of Virtual Memory . . . 8

II Group Milestones 9

2 Milestone 2: Processes, threads, and dispatch 10
2.1 Creating a process . . . 10
2.1.1 Setup initial C- and V-Space . . . 10
2.1.2 Setup the dispatcher . . . 12
2.1.3 Start the process . . . 12
2.2 Extending our memory management . . . 12
2.2.1 Managing the virtual address space . . . 12
2.2.2 Paging arbitrary memory . . . 13
2.2.3 Unmapping . . . 15

3 Milestone 3: Message passing 16
3.1 RPC . . . 16
3.1.1 LMP Library . . . 16
3.1.2 Channel Setup . . . 18
3.1.3 RPC Messages . . . 19

3.1.4 Supported RPC Operations 20
3.2 Monitor Server 22
3.2.1 Process ID 22
3.2.2 Server-Side RPC 23
3.2.3 Performance implication 23
3.2.4 Spawn Library Update 23
3.3 Team001 Comparison 24
3.3.1 General Server Design 24
3.3.2 RPC Design and LMP/UMP Switching 25

4 Milestone 4: Page fault handling **26**
4.1 Handling pagefaults 26
4.2 Morecore and Garbage Collection 27
4.3 Demand Paging of Thread Stacks While Disabled 28
4.4 Handling concurrency 29
4.5 Reentrant Paging Calls 29

5 Milestone 5: Multicore **30**
5.1 Booting a secondary core 30
5.1.1 Create KCB 31
5.1.2 Load Boot & CPU driver 31
5.1.3 Coredata 31
5.1.4 Clean the cache 32
5.2 Multicore Memory Management 32
5.3 Inter-Core Communication 33
5.3.1 Encapsulate RPC Messages 34
5.3.2 Transmitting the Core Bootinfo 34
5.3.3 Barriers 35

6 Milestone 6: User-Level Message Passing **36**
6.1 Message Passing Design Evolution 36
6.1.1 Overview 36
6.1.2 Message Passing Layer 37
6.2 URPC Message Passing 40
6.3 Duplex RPC Channels 42
6.4 Deadlocks in Single-Thread Message Handlers 43
6.5 Allocating Memory while Receiving a Response 44
6.6 Monitor Server Extension 45
6.6.1 Refactoring the Bookkeeping 45
6.6.2 Server-Side RPC 46
6.6.3 Performance implication 47

III Individual Milestones **48**

7 Milestone 7: Individual projects: building on the core **49**

8 Shell **50**
8.1 User-Space Console I/O 50
8.1.1 User Space Device Drivers 50
8.1.2 Interrupt-Based Character Reading 51
8.1.3 Multiplexing 51
8.1.4 RPC Message Integration 54
8.2 Shell Core Functionality 56
8.2.1 General Design 56
8.2.2 Parsing 59
8.2.3 Control Characters 61
8.2.4 Built-In Commands 61

9 Filesystem **63**
9.1 General Design of the Filesystem 63
9.1.1 Filesystem Layout 63
9.1.2 Improvements 64
9.2 SDHC Interface 65
9.2.1 Setup 65
9.2.2 Read/Write operations 65
9.3 FAT32 Library Design 66
9.3.1 Filesystem Type 66
9.3.2 Setup 68
9.3.3 Opening Files 68
9.3.4 Creating Entries 69
9.3.5 Updating Entries 69
9.3.6 Deleting Entries 70
9.3.7 Loading ELF Binaries 71
9.3.8 Improvements 71

10 Networking **73**
10.1 ENET driver 74
10.1.1 TX Queue 74
10.2 Packet Processing 75
10.2.1 Ethernet Layer 76
10.2.2 ARP Layer 77
10.2.3 IP Layer 77
10.2.4 ICMP Layer 78
10.2.5 UDP Layer 79
10.3 Provided Services 79
10.3.1 PING command 79

10.3.2 Register to listen on a UDP port . . . 81
10.3.3 Send UDP data . . . 81
10.4 UDP Echo Server . . . 81
10.5 ENET Driver Performance . . . 82

11 Nameserver **83**
11.1 Nameserver Process . . . 83
11.2 Bootstrap . . . 84
11.3 Nameserver API . . . 84
11.3.1 Server API . . . 84
11.3.2 Client API . . . 85
11.4 Service Routing . . . 85
11.5 Using the Nameserver in our System . . . 86
11.6 Waiting Until Service is Available . . . 86

12 Capabilities revisited **88**
12.1 Sharing memory . . . 88
12.2 Capability operations . . . 89
12.2.1 Delete . . . 90
12.2.2 Revoke . . . 90
12.2.3 Retype . . . 90

IV Our OS **92**

13 Performance evaluation **93**
13.1 Networking . . . 93
13.1.1 Upstream Bandwidth . . . 93
13.2 User Space Latency . . . 95
13.3 Message Passing . . . 97
13.4 Filesystem . . . 100
13.4.1 SD Block Driver . . . 100
13.4.2 File Operations . . . 101

14 Testing **102**
14.1 The unit testing framework . . . 102
14.2 Testing modules on top of the core . . . 103

References **104**

Appendices **105**

A User Guide **106**
A.1 Network Stack Interaction . . . 107
A.2 Shell . . . 108

A.2.1 Pitfalls 108
A.2.2 Command Help Pages 108

Part I

Individual Early Milestones

Chapter 1

Milestone 1: Memory Management and Capabilities

1.1 Physical Memory Management

In this milestone we implement `libmm`, a library to manage and allocate physical memory for the rest of the system. When starting up, the init process collects all available RAM-caps from the `bootinfo` struct and registers them with our library through `mm_add`. We call these large RAM-caps regions.

On a high level, the memory manager keeps track of each chunk of memory and uses it to serve allocation requests through `mm_alloc` and `mm_alloc_aligned` for aligned allocations. Any memory allocated this way can be freed using `mm_free`.

Internally, `libmm` keeps track of its memory with `mmnodes`. Listing 1 shows the layout of such an `mmnode`.

```
struct mmnode {
    enum nodetype type;
    genpaddr_t base;
    gensize_t size;
    struct mmnode *prev;
    struct mmnode *next;
    struct mmnode *prev_free;
    struct mmnode *next_free;
    struct regioninfo *region;
};
```

Listing 1: Layout of a node in the memory manager.

The `type` determines whether the node is free or allocated and `base` and `size` define the physical address range of this node. The `region` pointer contains information (`capref`, base, size) of the region added through `mm_add` that this node belongs to.

We maintain two lists of nodes. One list that contains all nodes regardless of allocation state and one containing only the free nodes for faster allocation. The `next` and `prev` pointers are used for the former and build a doubly linked list. The `next_free` and `prev_free` pointers are used in the free list and build a circular doubly-linked list. The `next_free` and `prev_free` pointers are `NULL` if and only if the node is not free.

The list containing all nodes maintains a few invariants, which hold whenever the list is in a consistent state (in between calls to the library).

- All nodes belonging to the same region are next to each other.
- Any two adjacent nodes of the same region form a contiguous memory range. This means that adding the size of a node to its base will result in the base of the next node.
- No two adjacent nodes of the same region are both free.

This allows for constant-time coalescing of adjacent free nodes and ensures that nodes are always coalesced.

1.1.1 Allocation

Whenever an allocation is requested, we search the free list for a node that can fit a block of the requested size and alignment. This node is split up so that we get a node that is exactly the right size that we can allocate.

We require that the requested alignment is a power of two. This helps decrease the complexity of the allocator.

We retype the RAM-cap for the region the selected node belongs to into a smaller RAM-cap for exactly the range of the selected node. This capability is returned to the user. New capabilities are created on-demand for the memory regions a user allocates, non allocated nodes do not have ram caps associated with them.

We use next fit for allocation by maintaining a pointer to the first free node after the previously allocated node.

Splitting Nodes

If the selected node is larger than the desired size, we split it up to avoid internal fragmentation.

Each allocation has a requested alignment, if the base of the selected node is not aligned correctly, we first have to cut off bytes at the beginning to get to the right alignment. This results in an extra node.

Additionally, if there is extra space at the end, we also split that off into an extra node.

Each allocation results in at most two new nodes.

1.1.2 Freeing

Freeing a node requires us to search the list of all nodes for the correct node. When freeing, the node is added to the free list and we try to coalesce it with its left and right neighbor, if they belong to the same region.

In contrast with traditional `malloc`/`free` implementations, freeing here is a relatively expensive operation because the node metadata is not stored right next to the memory the node represents, but in a separate list. This means freeing potentially requires walking the entire list of nodes.

The only way to identify the right block is by the base address passed to `mm_free`, so to reduce the lookup time we would need a faster way to find a node with a given base address. Since we use linked lists, we cannot use binary search, so to do lookup in logarithmic time, we would need to use a tree structure for the nodes or a dynamically growing array.

If the performance of `mm_free` becomes a problem, we will need to consider one of these approaches. But for this milestone we cannot say yet because `mm_free` isn't used anywhere.

1.1.3 Slot- and Slab-allocator

To maintain our internal data structures, we use a slab allocator for allocating the `mmnode`s (and the `regioninfo` structs). We also use a simple slot allocator for allocating a capability slot for the capability we return when allocating memory.

Since both these allocators need physical memory to allocate from, we have to be careful with how they allocate physical memory as they might try to recursively allocate physical memory when we call them to allocate a slab or slot while already serving an allocation request.

There are two ways to solve this: make the library re-entrant or make sure the recursion does not happen. We opted for the latter. We deliberately refill both allocators after our data structures are in a consistent state (at which point recursive allocation is fine) and the refill functions for both allocators detect when they themselves try to recursively refill themselves and ignore the recursive refill request.

For the slab allocator we only refill if the number of free slabs drops below 32. This is enough to allocate new memory for the slab allocator without running out of slabs in the process. In the slot allocator, we can always call the refill function as it only allocates a new CNode once it has used up an entire CNode.

Initially the slab allocator is backed with a static buffer to be able to get the memory manager running. The slot allocator also has an entire empty L2CNode provided by the kernel that it can use initially.

1.1.4 Minimal Block Size and Alignment

Unlike `malloc`, `libmm` does not need to be optimized for arbitrary allocations. Most allocations will be 4096 bytes or more, with the current `morecore` implementation always allocating multiples of the page size.

For this reason, we set the block size of our implementation to 4KB. Each memory region that is allocated is a multiple of the block size and will be aligned to a multiple of the block size.

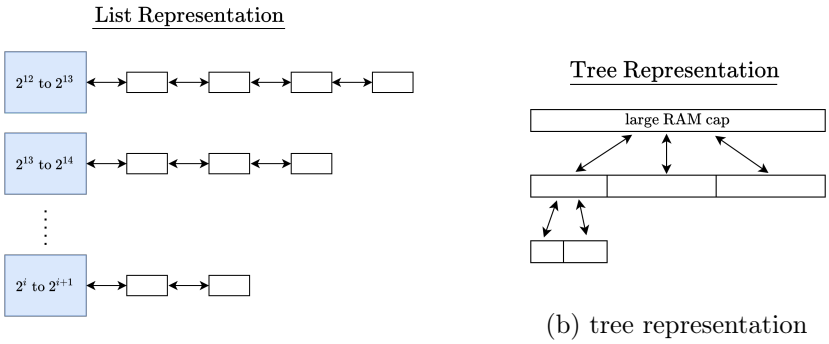
This allows us to place strong a invariant on our list of nodes, which in turn give us the following nice property:

- Each `mmnode` contains a memory region that is a multiple of the block size and is aligned to a multiple of the block size.

This means that any allocation that is a multiple of the block size will not have any internal fragmentation. In addition, as long as there are free nodes, a 4KB request with at most 4KB alignment can always be satisfied.

In theory, the block size could be changed to any other power of two, to optimize the allocator for different use-cases. However, in practice `cap_retype` requires that the offset is 4KB aligned, this makes it difficult to allocate less than 4KB without creating fragmentation. For example, if we allocate 1KB, we can have a 4KB aligned offset, however the 3KB after the allocated memory can never be retyped, because their offsets are not 4KB aligned. This could be solved by retyping the 4KB block of memory into four 1KB caps at once, however, this would then require keeping track of those extra caps separately, so we opted to keep it at 4KB.

This does have the drawback that any smaller allocations waste memory because of the internal fragmentation. However, so far this has not been an issue.



(a) list representation

(b) tree representation

Figure 1.1: Alternative design evaluated by Team001.

1.1.5 Team001 Comparison: Alternative Memory Manager Design Evaluation

Since one of our team members was in Team001 for the first three milestones, we occasionally add a short comparison to other design choices when there is an interesting difference to our Team's approach.

For `libmm`, Team001 evaluated a more complex design, which maintains lists of `mmnode`s using two different data structures.

First, they used a doubly linked list of `mmnode`s for physical memory of a particular size. For this purpose, they used buckets for sizes that are in between power-of-two bounds, starting at the base page size. The i-th bucket contains all nodes of size X, where $2^{11+i} \leq X < 2^{12+i}$. The sizes of the list go up to the largest region that was added by `mm_add`. Figure 1.1a visualizes this data structure. We maintain two different sets of those lists, one for free `mmnode`s and one for allocated ones.

Second, for each memory capability, they also maintain a tree data structure shown in Figure 1.1b. Every node tracks a list of its descendants, which are nodes that were split off from the parent RAM cap. Each node, except the first RAM caps, keeps a pointer to its parent node, from which it was created. When a node is freed, the tree is used to free other nodes. First, all `mmnode`s created from the freed node are found and freed by recursively going through a node's descendants list. Second, they traverse the tree upwards and coalesce free nodes to a larger free region whenever all descendants of a node are free. The first regions, added with `mm_add`, are detected through a `NULL` parent pointer and never removed.

In comparison to our doubly-linked list approach, this design has the following advantages:

- **Fast free-block lookup:** since lists are ordered by their sizes, they can quickly find a block of size X, by first going to bucket i of blocks of size $2^{11+i} \leq X < 2^{12+i}$ and then continue to go through this bucket's list until they find a block that is not only large enough but also satisfies the alignment. If none is found, they continue with bucket $i + 1$. This is faster, because they only traverse free blocks that are larger than X.
- **Fast allocated-block lookup:** they can find an allocated block in logarithmic time by traversing the tree from the large region that contains the block. Especially on long-running or fully utilized systems, there are few large regions but many smaller blocks. Thus, it is easy to find the large parent caps from which the searched block was split off. However, this was not yet implemented.

Significant disadvantages of Team001's alternative design are:

- **Complexity:** clearly, the performance improvements come at the cost of increased complexity. This design was harder to test for correctness. Usually, bugs are also more time-intensive to resolve for a more complex design. Those were the main reason why Team001 abandoned this design in the end and picked one similar to our team's doubly-linked list.
- **Overhead:** maintaining two different data structures at the same time is not only complex, but also leads to more operations. Team001 benchmarked this implementation against a one based on doubly-linked lists and for the tested access patterns, the complex approach did not show significant advantage.
- **Predictability:** the complex approach turned out to have surprisingly unpredictable performance. One reason for this is delayed coalescing: if they always allocate small `mmnode`s of the same size, they create a deep tree structure and defer coalescing until the lowest child is freed. This might trigger a large number of expensive cap destroy operations. At the same time, freeing other `mmnode`s is very cheap, because they are neither coalesced nor is their cap destroyed. We are sceptical that such unpredictability at a low level of a system is a good design trade-off.

Both designs have fragmentation. Team001's design tends to keep smaller `mmnode`s around, because not all of the parent's descendants are freed. On the upside, this means that plenty of small nodes are already available and can readily be reused for a request of the same size. On the downside, it is possible that they cannot allocate a larger region despite enough free consecutive memory, due to more complex access patterns that lead to adjacent `mmnode`s that are free, but not coalesced, because they have allocated siblings. In comparison, our doubly-linked list approach can always coalesce adjacent free `mmnode`s. However, depending on the access pattern, it more often splits

larger memory regions into smaller ones, because it eagerly coalesces them as soon as it can instead of reusing them.

In hindsight, choosing a simpler memory manager was a good decision. As the project continued, we discovered bugs in the `libmm` implementation (for both Team001 and our team) that were hard enough to debug even with a simpler memory manager.

1.2 Mapping of Virtual Memory

For this milestone, we only implemented a proof of concept virtual memory manager, which would be replaced almost immediately in milestone 2. The paging state contained fixed mappings for the L1 and L2 page tables as well as space for all 512 L3 page tables in the fixed L2 table. On the first call to `paging_map_fixed_attr`, we lazily allocate and mapped the L1, L2, and L3 page tables for the requested address using a boolean flag to keep track of which L3 page tables were already mapped.

This memory manager was already capable of mapping up to 1GB of memory, with some restrictions:

- No mapping request could touch multiple L3 page tables.
- All virtual addresses must lie in the same L1 and L2 page table.
- All mapping requests must map a multiple of the page size.

Part II

Group Milestones

Chapter 2

Milestone 2: Processes, threads, and dispatch

2.1 Creating a process

In preparation to start a new process there are a few steps required. The relevant ELF image for the program has to be found and parsed correctly. In addition we have to setup the initial CSpace and VSpace for the new process and tell the CPU driver about the new dispatcher in order to be successfully scheduled and dispatched. These steps are described in more detail in the following chapter.

2.1.1 Setup initial C- and V-Space

Before starting, a new process expects certain capabilities in its capability space and certain mappings in its virtual address space. Our spawn code is responsible for setting this up.

C-Space

Capabilities for the child process have to be placed at fixed position in its C-Space. These capabilities include, among others, the capability for the root CNode itself, the L0 page table, the frame-cap for the commandline arguments and the dispatcher frame. For the complete list and concrete placements, we refer to the AOS book [1, Section 4.8]. The initial C-Space also includes empty slots for a slot allocator to use and empty slots to store capabilities for pagetables and mappings, as well as an L2Cnode filled with RAM-caps, at this point, none of them are used for spawning the process.

When spawning the process, we create the root CNode for the process as well as the L2Cnodes for the capabilities and free slots mentioned above. Finally,

we copy all the capabilities into the expected slots in the child's C-Space.

V-Space

Initially setting up the virtual address space only requires us to allocate a vnode for the L0 pagetable and then passing that together with an uninitialized paging state (the child's paging state) to the `paging_init_state_foreign` function. After that, anytime we need to map something into the child's V-Space, we can use the child's paging state in any of the paging functions.

While spawning, we need to map certain frames into the child's address space. In particular, we map the frame containing the process' commandline arguments and environment and the dispatcher frame. In addition, we map each ELF section at a fixed address given by the ELF binary in the `elf_allocator` function which is called as a callback from `elf_load`.

For this milestone, we don't share the paging state we created while spawning the child with the child process once it starts. This means the child must not try to map any pages or page tables that were already mapped during process creation. The child process is given `VADDR_OFFSET` (512GB) as the base address of the virtual address space it has access to. `VADDR_OFFSET` is chosen in such a way that any address below it will go through slot 0 of the L0 page table, while all other addresses will use higher slot numbers. If that wasn't the case, the child might try to map into a slot in the L0 page table that was already mapped when creating the process.

To avoid any conflict, we have to make sure that nothing mapped in our spawn code is mapped at or above `VADDR_OFFSET`. Because of that, we use `BASE_PAGE_SIZE` as the base address of the V-Space in `paging_init_state_foreign` when spawning the process. We cannot use 0 as the base, because then the paging code could make the NULL address a valid memory location.

The argument and dispatcher frames are mapped using `paging_map_frame`, this function simply allocates and maps the first free virtual memory address. So as long as we don't map more than `VADDR_OFFSET` this way, we will not exceed our upper bound. The ELF sections are mapped to addresses given in the ELF binary, which we have no control over. In theory these could be mapped above `VADDR_OFFSET`, which is why the `elf_allocator` returns an error if this would occur.

If this becomes a problem, one solution would be to share the paging state we setup during process creation with the child process so that it has a complete view of all mapped pages and pagetables when starting up.

2.1.2 Setup the dispatcher

During the V-Space setup we already mapped a dispatcher frame into the child's address space which is used to store all information about a process. The CPU driver will use this memory (e.g. at a context switch) to store information about the process, but for the first switch some information has to be set in `dispatcher_shared_generic` beforehand.

- The virtual address of the dispatcher frame in the child's VSpace
- Start in disabled mode (used for scheduler activations)
- Entry point where it should start executing (provided by the ELF)
- Base address of the `.got` in the child's VSpace (provided by the ELF)

Setup arguments

In order to pass command line arguments, the initialization code uses the structure `spawn_domain_params` at the beginning of the argument page which was previously mapped into the child's address space. The child's startup code expects everything we haven't explicitly set to be zero. The structure contains arrays for the command line arguments and the environment strings, the end of the list is indicated by a `NULL` pointer. The actual arguments are copied into the argument page right after the structure `spawn_domain_params` with the corresponding pointers in `argv` pointing into the child's address space. The pointer to this described structure is passed using the register for the first argument in the enabled save area, which is then used to restore it during `thread_init_disabled`.

2.1.3 Start the process

After a lot of setup and memory mapping operations, we are finally ready to make our new dispatcher runnable by calling the dispatcher capability. This step requires the previously prepared capabilities such as the new dispatcher itself and the existing dispatcher, both relative to their own CSpace. We also need to pass the capability for the root of CSpace and VSpace and for the newly created dispatcher frame, the first relative to the parent's CSpace and the others relative to the CSpace of the new dispatcher.

2.2 Extending our memory management

2.2.1 Managing the virtual address space

To facilitate all the memory operations needed to spawn processes, first we needed to add some kind of support for managing our virtual address space. In the beginning we used a simple counter that started at a predefined offset

and just incremented a number for every address request. This worked great in the beginning, because it is simple (i.e., no bugs) and incredibly efficient. However, it came with a major flaw: there is no way to handle freeing regions of the address space (other than the end).

At the point when we needed this functionality we changed our implementation to use a new struct:

```
struct vm_node {
  lvaddr_t base;
  size_t size;
  bool is_free;
  struct vm_node *next;
  struct vm_node *prev;
};
```

This essentially formed a linked list, very similar to the one we use to manage our physical memory. The virtual memory allocation uses a "next best fit" strategy, where we start searching, from the last allocated node, for a large enough free node. Once found we split off a new node from the beginning to account for alignment and if the original node is too large we split off a new node that becomes the allocated node. This approach seemed reasonably efficient for our use case, since in the beginning we start with a huge node starting at the above mentioned predefined offset with the size of the whole virtual address space. For each incoming request for a virtual address we just need to split off 1 or 2 nodes from this huge node.

When a virtual address range is freed we just need to search for that node and mark it free. To avoid unnecessarily fragmenting our address space we also check if any neighbouring nodes are free and merge them if they are.

Pointers to both the list start and the next free node are stored in the paging_state struct. Memory for all nodes is allocated using a slab allocator that is initialized with a static buffer during initialization.

2.2.2 Paging arbitrary memory

Now that we can provide free virtual addresses we need to make sure a user can also map some memory at those locations. To track the state of the MMU we built a shadow page table:

```
struct page_table {
    uint64_t slot;              // Slot in the parent pt
    bool is_mapped;             // Is this entry mapped; false
    ↪  by default
    struct capref pt_cap;       // The pagetable capability
    struct capref map_cap;      // The mapping capability
```

```
    struct page_table *parent;  // The parent pt of this
    ↪  entry/pt
    struct page_table *entries; // An array of entries in this
    ↪  pt
};
```

Note the entries member. Every time a new page table entry is allocated/mapped the whole array of sub-entries is also allocated and this is potentially a large chunk of memory. One of the groups had to split up in the middle of the semester and our group got a new team member. The new member mentioned that they used an array of pointers instead. By using an array of pointers they would save some memory for every unmapped entry, compared to our representation.

I would argue that our approach performs slightly better for lookups since there is less indirection at every level. In the scenario, where memory accesses are dense (i.e. as continuous as possible) our approach can also be more memory efficient, since for every completely mapped entry we avoid the 512 element long array of pointers. In our case it is a 64 bit machine, so we could save $512 * 8 = 4096$ bytes (or a whole page). Since a page table is shaped like a tree there are exponentially more entries of lower level tables, which also have a higher chance of having all of their entries mapped.

However if memory accesses are sparse the approach of pointers would probably use noticeably less memory, depending on the amount of data. This discussion unexpectedly highlighted at least one benefit of user level paging, since even the efficiency of the shadow page table representation depends on the memory access pattern of the application using it.

When mapping a frame there were a few new considerations that we needed to take into account (or rather found out when testing). Initially we just created a page table walker that would traverse our shadow page table, creating entries/vnodes as needed and returned the L3, where we just mapped the frame we got and set the shadow entry to mapped.

However a frame might not always fit into the returned L3 table. It could be too big or simply be mapped right into the last slots. To handle these cases we map in a loop, where we map as much as we can/need in an iteration, then if needed increment the virtual address, re-walk the page table and start again.

After extending our memory management in our paging state we needed to have the slab allocators for the shadow page table entries and virtual address nodes as well as the slot allocator we use when mapping, the virtual address node list pointers for the start and the next free nodes and lastly the shadow page table itself. In code this looks like the following:

```
struct paging_state {
    struct vm_node *virt_head;
    struct vm_node *virt_next_free;
    struct slab_allocator vm_slabs;

    struct page_table root_table;
    struct slab_allocator pt_slabs;
    struct slot_allocator *slot_alloc;
};
```

2.2.3 Unmapping

To unmap a region of memory it is enough to lookup the page table entry using the given virtual memory address. From this entry the mapping capability can be retrieved and deleting this capability leads to the entry, being deleted from the underlying page tables that the kernel manages (or just generate page faults after that). After this the page table entry can be set as not mapped and the frame capability given back to `mm`, the memory manager.

The main issue here was, when we started mapping large frames that might have been broken up into several page table entry capabilities (and entries in the shadow table). Later, this turned out to not be an issue for demand paged memory, however we had one idea to create page table entry groups so that each member of the group could be retrieved, once one was unmapped. Sadly we didn't have time to implement this and became obsolete with time.

Chapter 3

Milestone 3: Message passing

3.1 RPC

3.1.1 LMP Library

From the start we knew that inter-process communication will have at least two mechanisms, depending on the core of the source and destination. For a process that simply wants to send some data over on a channel, this could be cumbersome, so in our implementation we created a wrapper library around LMP. This library would hide some of the complexities of using the LMP channel as well as an intended place to implement the same for the UMP channel and multiplexing between the two. Processes then could simply call this layer to send arbitrary messages over arbitrary channels.

In the beginning this library only implemented an asynchronous send and a synchronous receive (with and without capabilities):

```
errval_t mp_send(
    struct mp_channel *chan,
    const void *buf,
    uint8_t length
);
errval_t mp_send_cap(
    struct mp_channel *chan,
    const void *buf,
    uint8_t length,
    struct capref cap
);

errval_t mp_receive(
    struct mp_channel *channel,
```

```
    void *buf,
    uint8_t *retsize
);
errval_t mp_receive_cap(
    struct mp_channel *chan,
    void *buf,
    uint8_t *retsize,
    struct capref *cap
);
```

All of these simply sent/received messages on the given channel from/into a given buffer. The asynchronous send were easy to turn into synchronous ones by the marshalling layer if needed, by awaiting an "ACK" message and this worked great for 2 processes. The complexity of the event based receive was handled inside the library: it set up a callback function that copied the message into the given buffer and then waited for messages. This was nice, because the API is de-coupled from the API of LMP, giving us more confidence that the same API can be implemented on top of UMP, when the time comes.

However, a server (in our case `init`) needed to wait/receive for messages on multiple channels. So our choices were:

1. The LMP library accepts a set of channels for the receive. Registers it's handler for all of them. When a message arrives it de-registers the handlers for all other channels, handles the incoming message and returns the data to the caller.

2. The LMP library does not deal with the complexity of multiple channels and just provides an interface for registering callbacks.

Our team ended up implementing the second option. The first option sounds like a nice abstraction, where the message handler loop of the server receives a message, handles it and then receives the next, but is potentially a lot of complexity inside our library for not a lot of benefit. To implement the second option, we added the following new calls:

```
errval_t mp_register(
    struct mp_channel *chan,
    struct event_closure callback_func
);

errval_t mp_read_msg(
    struct mp_channel *chan,
    void *buf,
    struct capref *retcap
);
```

With these calls, users can register callback functions and read messages inside those function without directly interacting with LMP.

3.1.2 Channel Setup

In the beginning our channel creation was a two step process. Our `init` process had an endpoint that was passed along to all processes that were spawned. Each process upon spawning would send a message to `init` with its endpoint and a request for a separate channel. When `init` received this message it minted a new endpoint for itself, allocated a new channel and sent back the new endpoint capability. This "channel allocator" approach worked fairly well and seemed like a good protocol, when we start implementing servers, which might or might not be contacted by processes.

Unfortunately, when we started to spawn many processes at the same time, we ran into issues. With this approach it could occur that, while a processe's channel request is being handled a new process wants to contact `init`, but can't, because that endpoint has a full buffer. The process could wait until the buffer is cleared, but that could be an additional bottleneck in a system where lot's of processes are destroyed and created. Or some processes might not be able to start at all, because they never get the buffer, since other processes starve them. So we changed it, and that ended up working.

Each spawned process has a separate LMP channel to `init`, which is set up through a multi-step handshake protocol. When `init` spawns a new process, it first creates its LMP channel for the process, but without a remote endpoint yet. This channel contains a minted local endpoint capability as well as a 4KB shared frame used for passing larger messages (see Section 3.1.3).

To fully establish the channel, the child process needs to receive both `init`'s minted local endpoint capability as well as the frame cap to the shared frame. In addition, `init` needs a minted endpoint cap for the child process.

`init` passes its local endpoint to the child directly when spawning by placing it in `TASKCN_SLOT_INITEP` in the child's CSpace. The remaining capabilities are exchanged in the handshake protocol:

1. Once the child process starts, it initializes an LMP channel to `init` using the endpoint capability passed in its CSpace.
2. It then sends an `AOS_RPC_HANDSHAKE` handshake message together with the local endpoint cap for the channel.
3. `init` replies with another handshake message together with the frame cap for the shared buffer.

At this point both sides have a fully established LMP channel that can be used to send and receive arbitrary size messages.

3.1.3 RPC Messages

The LMP library was not intended for use by client programs, except for specialized circumstances (e.g. in `init`). All requests would go through our AOS RPC library. Clients would use the RPC library for sending specific requests like printing or allocating a ram cap.

The RPC library defines its own message format as follows:

```
struct aos_rpc_msg {
    uint8_t type;
    uint8_t data[31];
} __attribute__((__packed__));
```

The `type` field determines the meaning of the data bytes. The size of the struct was chosen so that it can be sent in a single call over LMP.

All messages sent in the system use this format, there is no lower layer message format or encapsulation. This was a deliberate choice because it allowed us to fully use all 32 bytes LMP allows us to send without sacrificing one or two bytes for something like an LMP header.

Due to how the buffers work in LMP, we needed to be sure that we only send an LMP message after the other side has read the previous one, otherwise we could have a buffer overflow. We achieved this by enforcing that every RPC request has a reply that we wait for when making the request. Together with locks around the send and receive functions, this ensured that no two messages could be sent on the same channel without first receiving a reply to one of them.

Large Messages

At first large messages were supported in the serialization layer. The sender would signal to the receiver that it has a large chunk of data it wants to transfer and asked for sufficiently large enough memory region. The receiver would acquire some memory, map it into its own address space and send back the capability for that memory. The original sender can then map this capability, copy the data, unmap and send a message telling the original receiver that the data is ready to read.

Unfortunately this was a "too manual" process of handling large messages and required too much duplication while marshalling messages, so a more automatic solution was needed. We wanted to keep the LMP library simple, so we did not want to handle such cases there.

Ultimately, we created one shared buffer per channel that could be used for sending larger messages. The shared buffer is 4KB, but this has no significance, it could just as well be any other size. The RPC library would

handle serializing and deserializing large messages and sending/receiving them over LMP. This is done using the special RPC message type `AOS_RPC_LARGE`:

```
struct aos_rpc_large {
    uint8_t type;
    char pad[7];
    size_t size;         ///< Number of bytes in this message
    size_t size_total;   ///< Number of bytes in all message
    ↪  parts combined.
} __attribute__((__packed__));
```

If the sender notices that the message it wants to send is too large (more than 32 bytes), it starts a large message transfer. It does this by writing as much message data as possible into the shared buffer (up to 4KB) and creates an `aos_rpc_large` message that lets the receiver know how much data is contained in the buffer and how much total data we intend to send. It then sends this message over LMP where the receiver now can start reading from the shared buffer and reconstruct the message. Each message has to be acknowledged because sending two subsequent LMP messages leads to a buffer overflow.

In most cases large messages can be sent with a single transfer through the shared buffer. But if the message is larger than 4KB, multiple rounds are needed. The process is exactly the same: We send as many `aos_rpc_large` messages as needed and fill up the shared buffer each time until there is no data left to read.

The receiver knows from the beginning exactly how much data to expect and can already allocate a buffer for it.

3.1.4 Supported RPC Operations

Requesting RAM Capabilities

Because LMP allows us to send and receive capabilities, a process can request RAM caps from a memory server and use them in whatever way it likes. In our implementation, the memory server runs directly in the init domain because it makes things a lot easier and the memory manager also already runs there.

Requesting a RAM cap is relatively straight forward:

1. The process calls `aos_rpc_get_ram_cap`, which is basically a proxy for `mm_alloc_aligned`.
2. The RPC library sends an RPC message of type `AOS_RPC_RAM` over the LMP channel to `init`.

3. The message handler in `init` is triggered and it allocates and sends back a ramcap
4. After the reply is sent, `init` destroys the ramcap.

The last point is important because otherwise there is a stray copy of a capability in the system which both leaks memory and may prevent the memory range to be reused if the memory is ever given back.

Managing Processes

See Section 3.2.2.

Input / Output

Starting with this milestone, printing to standard out should run through a centralized serial server. In our case, again everything runs on `init`. The existing code of barrelfish allows us to overwrite the low level functions used for writing to standard out (and reading from standard in). Here we fall back to going through the kernel for printing if no rpc channel is available. If one is available, we send an RPC message to `init` that contains the string to be printed, which init will then print for us.

Similarly for input we can send rpc messages to init, which in turn can use function like `getchar()` to read input on a character by character basis. In the clients/processes we wanted to support arbitrary input function, so for every character there is an rpc message. This works quite well if there is no other IO activity, but quickly runs into issues otherwise.

First, if other processes want to print, while we are getting input, after each keystroke a chunk of text is going to appear in the time between two character requests. Second, if other processes want to read characters too, the characters are going to get mixed.

To solve this we introduced an ownership/queue based locking system. The serial server (`init`), provided an interface to lock and unlock stdin. If there was no owner `init` responded with an ACK, otherwise the request was added to a queue and the requesting process instructed to wait until notification (a message signaling that it had been granted ownership). When stdin was locked each print request had to wait until stdin was unlocked to avoid the characters written getting lost (and the weird block of text appearing after each keystroke).

This worked reasonably well for a single core, but this was no longer possible with multiple cores. For one, if core 1 called `getchar` but there was already a lock on stdin, the entire UMP channel between the cores would block, stopping

all message transfers from core 1 to core 0. This system was completely reworked as part of the shell in Chapter 8.

3.2 Monitor Server

Beside the memory server we also have a monitor server running on the `init` domain. Other domains can now use the client-side RPC functions to create a new process, retrieve a list of process IDs of all running processes and also get the corresponding name of a given process ID if it is running. Therefore our monitor is not just responsible for spawning a new process, it also manages all the spawned processes and gets notified if a running process exits. Listing 2 shows the layout of such a `monitor`.

```
struct monitor {
  enum processtype type;
  char *binary_name;
  domainid_t pid;
  struct mp_channel *mp_chan;
  struct spawninfo *si;
  struct monitor *next;
  struct monitor *prev;
};
```

Listing 2: Layout of a monitor structure.

The `next` and `prev` pointers are used to maintain a doubly linked list of spawned processes. Each list element corresponds to one specific execution of a program spawned by the monitor and contains additional information such as the `spawninfo` built during the actual spawning in `libspawn`. It also contains an `mp_channel` that has been assigned and registered to the `msg_handler` event handler.

3.2.1 Process ID

We assign a unique process ID (PID) for each spawned process, starting at 0 for the `init` domain. A static `domainid_t` is used to keep track of the `next_pid` and each time a new process needs a PID, we test the `next_pid` against the monitor list to see if the PID is already in use. If not, we assign it to the process, otherwise we recursively try the next one. Currently we use 32768 as the maximum PID and start again at 1 after we reach that point. Our current implementation tries to obtain a free process ID in an infinite way, assuming that our system will not have all available process IDs in use at the same time for now.

3.2.2 Server-Side RPC

The client-side RPC functions send messages that trigger the correct server-side RPC functions in the monitor via our new LMP infrastructure, which then receive a response back. This includes handlers for spawning, retrieving all PIDs, retrieving the name of a corresponding PID and when a process is terminated. All those handlers operate on the linked list structure by adding a new element to it when spawning, iterate through the list to get all PIDs or to return the name of a matching PID and remove elements from the list when exiting. All those server-side RPC functions can be called freely except the exit one, this is called during `libc_exit` to notify our monitor about the event to keep the list up to date and make the used process ID available again. In order to pass the right PID to `aos_rpc_process_exit_pid` we use the `domain_id` from the corresponding dispatcher, which is set to the assigned process ID in `setup_new_dispatcher` during the spawning code. This way the monitor gets notified about the exiting process and can safely remove its structure from the list.

3.2.3 Performance implication

Managing our spawned domains as a linked list has the performance advantage when retrieving a list of all process IDs which is constant in the number of processes in our list. But every other operation has the disadvantage that it depends performance-wise on the number of processes that are currently being managed by our monitor. Since retrieving a list of all PIDs is not an operation that is used very often, we refactored the basic structure described in chapter Chapter 6 to avoid performance dependency in our monitor code and the number of processes managed by our monitor.

3.2.4 Spawn Library Update

Adding a monitor has also changed the way our `init` domain creates new processes. The provided functions in the spawn library now act as a wrapper that forwards the spawn request to our monitor. It is important to note that the supplied `spawninfo` reference passed to the wrapper is ignored, while the process ID is actually set by the monitor. The reason for this is that our monitor does the bookkeeping for a new process and when we spawn a new process we already set some parts of the new `spawninfo` for the process in the monitor itself, such as the `capref` for the `init` endpoint. Most of the setup in the monitor was added in later milestones, but we mention it here already because of the drastic change in the spawn library. So when the monitor receives a spawn request, we call the spawn library again, where the actual spawn setup described in the previous chapter now takes place.

3.3 Team001 Comparison

3.3.1 General Server Design

Team001 decided to implement all servers in their own processes instead of handling them directly in the `init` process. The goal of this design was to avoid making `init` the bottleneck for all requests.

This introduced a lot of complexity for M3, because this decision entails that one has to set up LMP channels to all servers and between the servers. For this purpose, Team001 introduced a single receive endpoint for the servers and mapped this to the CSpace of every child process. They added a special RPC message, transparently hidden from the caller, that sends a connection request on the first use of an RPC channel. This request contains the sender's receive endpoint as well as its domain ID. The servers use a "channel manager" data structure to keep track of LMP channels for domain IDs. Each subsequent request to the server contains the source domain ID and goes to the server's single receive endpoint. The server then sends its response to the LMP channel registered for this domain ID. An additional complexity was that we needed to initialize a second memory manager and provision it with RAM capabilities. Our team only faced this issue in the M5 when we transitioned to a multi-core system (see Section 5.2). Overall, getting this to work for M3 turned out to be more challenging than anticipated and was one of the reasons why Team001 had to late submit M3.

In hindsight, it was too ambitious to try to address challenges that only occur later. The advantage of only dealing with such challenges when they cause problems, is that you have a better understanding of the system at that point: the entire system is more evolved, you spent more time implementing parts of it, and there is a concrete issue to solve. Together, this makes it easier to find a good solution for the problem. For example, during M3 we did not have multiple cores nor did we use many threads on a single core. Thus, handling all servers in `init` is not an issue, because only one thread can be scheduled on the core anyway. However, centralizing everything in `init` reduces the complexity significantly. Later, it turned out that we need to handle requests in different threads, because they might block waiting for a response from another thread. When we only use a single `init` thread, the thread we wait for cannot use any of the servers, since `init` is blocked. However, this issue is not resolved by using different processes for each server (as Team001's plan was), this just reduces the issue: now, the waited-for thread cannot use the same server that is waiting for its response. Thus, we still need multiple threads for each server! But then, we can just as well always use different threads for handling messages in `init`. This is the solution our team adopted once this issue occurred, as we describe in Section 6.4. This still has the advantage of having a much simpler setup. It also centralizes message

handling, making it easier to maintain and reducing code duplication. In conclusion, we were able to come up with a better solution, because we had a better understanding of the problem later in the project.

3.3.2 RPC Design and LMP/UMP Switching

Both teams built an LMP abstraction layer for using LMP channels. However, Team001 designed it to enable sending arbitrary size data, while our team decided to limit it to send 32-byte messages and then built arbitrary message partitioning inside RPC.

Team001 again tried to anticipate the introduction of UMP channels and gave the RPC struct a type, defining which channel it is underneath. Their goal was to transparently switch inside the RPC implementation between using the LMP respectively UMP abstraction layer for sending the data. In hindsight, this would have caused some code duplication, since the RPC channel sends the same messages over both channels. In our team, we later implemented an additional abstraction layer between RPC and LMP/UMP (called MP, see Section 6.1). Thus, we pushed down switching between the layers so that RPC can use unified message passing channels and they internally deal with the constraints of the different channels. We intended to avoid duplicating code by implementing arbitrary size message sending in the MP layer and reuse the code handling packet fragmentation for both LMP and UMP layers. However, this turned out to conflict with avoiding unnecessary message copying, so we ended up having a lot of LMP/UMP code branching in the MP layer. In the end, both the approach of thin LMP/UMP abstractions for fixed-size messages and the other approach of more sophisticated, arbitrary-size message-sending LMP/UMP abstractions seem to be viable design choices. They just lead to code duplication at different places, at least when we try to minimize the number of `memcpy`s.

Chapter 4

Milestone 4: Page fault handling

4.1 Handling pagefaults

Barrelfish provides a convenient way for user space programs to handle pagefaults allowing them to decide on how to manage their memory. We have used this functionality beforehand for our testing code in order to verify that there occurred a pagefault, where we expected one and to not rely on the test suite pagefaulting when we verified that some of our code does not trigger a pagefault.

During the initialization phase of our paging code a new handler is registered with a static buffer for the exception stack. This handler facilitates demand paging, which has the following steps:

1. Verify the exception (e.g. the address is in the expected range)
2. Acquire physical memory
3. Determine the mapping base and map to that address the new physical memory

This would be a really memory efficient system for use-cases, where we don't know how much of our allocated memory is going to be used e.g. an input buffer for strings. However, our team quickly found out that if all of the memory allocated is accessed this approach requires at least one exception, one physical memory allocation call and one system call to map for every 4 KiB. Our solution for this was to trade space for time and for every exception fill up a whole level 3 page table (2 MiB). It is slightly more inefficient in terms of memory, but the performance difference of reading/writing a whole 100 MiB area was very visible. (one of our test cases did that

In theory we could have implemented this by terminating our page table walker code sooner, which was written with this in mind. Unfortunately we were under time pressure and encountered buggy behaviour and had to resort to using the pte_count argument of the vnode_map function.

4.2 Morecore and Garbage Collection

Morecore is backing the memory regions malloc would use, therefore we needed to implement a sort of connector for our paging code. This handles operations like initialization, allocation and freeing. The state of morecore is tracked in `struct morecore_state` to which we added a single vm_node, which represents a range of virtual memory addresses allocated by paging, to the heap. This is allocated during initialization and is set to a hard-coded huge value (on the Toradex board, there is a very small chance of running out of virtual memory regions unless, they are never freed).

To free memory on the heap, during the handling of every `free()` call another function `lesscore()` is called. This call is meant to facilitate garbage collection or reclaiming resources from malloc. We have implemented a garbage collection striving for maximum memory efficiency. Our understanding was that malloc keeps a sort of circular list of free pointers. Each pointer is a header at the start of each block that malloc "owns" and each header contains a pointer to the next block, a magic number that signals if the area is free or allocated and the size (where the size is stored in terms of units that are as large as the header, which in turn is a union aligned to a long long for maximum fun).

During garbage collection we iterate through the list of empty blocks. If a block is at the end of the heap, it's great we can simply unmap it, decrease the heap and we are done. If not there are still things we can do. We cannot unmap the whole free area, since it also stores the bookkeeping information at the front, but we could unmap everything after the first page still potentially reclaiming some memory. So our requirement is that it is larger than 4 KiB and the we try to unmap it. In these cases we also modified the magic number of the malloc header, so we can check in a subsequent garbage collection pass if it was already unmapped.

Since the pages we want unmap were mapped as 4 KiB or 2 MiB regions of memory (with 2 MiB regions aligned to start at slot 0 of their L3), we didn't have to worry about larger frames splitting up and having more than entries in the shadow page table. This meant that we could simply walk the page table, get the mapping capability, delete it and set that entry as not mapped.

This worked nicely for a while, however in a later milestone our `lesscore()` implementation introduced a particularly hard to solve issue. In some cases

the garbage collector would free some memory regions that contained important data (like our rpc struct). This was hard to reliably reproduce in some case, which did not give us too much confidence in our fixes. Garbage collection was not a requirement or challenge to implement and we were worried about future time and effort needed to maintain this piece of code we disabled it.

4.3 Demand Paging of Thread Stacks While Disabled

We found this issue only in the final days of the course, but thought it would best fit here.

For every new thread created, a fixed-size stack is allocated using `malloc`. As with all other memory allocated this way, as soon as an unmapped page was touched, our page fault handler dynamically allocated and mapped physical memory for it.

This has one major flaw: What if this happens while the dispatcher is disabled?

In that case, the page fault is handled by the kernel and may crash the process.

This means we need to make sure that any memory touched while disabled cannot trigger demand paging. So far we have only found this happening with the thread stack, but it's difficult to know because this is not easily reproducible.

With the thread stacks, we solved this by using `calloc` to allocate the stack which touches the entire stack, thus ensuring that there will be no demand paging on it (unless there is a stackoverflow). While this does make threads a bit more heavyweight, it is the only proper solution we could come up with.

However, this leads to a broader issue: All code that one writes that can run while the process is disabled, must not touch malloc'd memory for the first time.

The idea behind demand paging was that it would transparently allocate memory once it is accessed, without the user noticing. But this requirement makes this hidden "feature" implicitly part of the API of all functions that disable the dispatcher.

In addition, the stack is one of the only allocated memory regions where you could expect that parts are never accessed. In most cases, one allocates memory because one expects to access every single byte of it.

This makes us question whether demand paging is really worth it.

4.4 Handling concurrency

In this milestone we were also tasked to test out implementations with threads. Threads share a single address space, consequently a single page table, which could lead to a race condition, where two or more threads fault inside the same area. This could lead to double mapping, memory leaks, and lost data for some threads. The solution to this was fairly simple. We protected the paging inside the page fault handler with a lock and in the critical section, we first checked if the faulting address was still not mapped. If it was mapped, it meant that it was mapped while we were already in the pagefault handler.

4.5 Reentrant Paging Calls

Mapping pages, may require us to allocate higher level page tables and slots for the mapping capabilities for those VNodes during a page table walk. During both of these allocations we may have to allocate memory through `libmm` and `libmm` could refill the slab allocator. This causes a recursive call to the paging library which may result in a page walk that touches and allocates some of the same L1, L2, or L3 page tables.

If that is the case, after allocating a page table, we need to check if it was not already mapped recursively. In that case, the allocated page table needs to be deleted again and we can normally continue our page walk. The same thing also has to be done after allocating a slot for the mapping capability of that page table.

These recursive paging calls can never designate the same memory region for two paging calls, because we reserve the memory range before doing any allocations.

Chapter 5

Milestone 5: Multicore

5.1 Booting a secondary core

To start a new core, we need to invoke the inter-process interrupt (IPI) capability, which is only available to privileged user-space processes like our `init` process. A platform-specific boot protocol starts the boot driver which afterwards loads the CPU driver on the newly booted core. While all this is done by the System Control Unit (SCU) we have to prepare the new CPU driver. This includes a lot of setup and allocating memory beforehand:

- **Create KCB:** The Kernel Control Block holding the state for the CPU driver.
- **Load Boot & CPU driver:** The boot driver prepares the environment (e.g. setup a default kernel page table) and jumps into the CPU driver.
- **Coredata:** Provides the boot parameters to the new CPU driver.
- **Kernel stack:** Stack for the new CPU driver.
- **Prepare initial task (`init`):** Allocate memory used for the allocations of the new CPU driver and space to actually load the `init` process. In addition, we need to provide a memory area where the CPU driver expects the ELF image of the `init` process.
- **Clean the cache:** Make sure everything it needs is visible and sitting in RAM.
- **Call spawn:** The PSCI function `CPU_ON`, which starts a new core, calls the System Control Unit, but only EL1 and higher are allowed to call the SCU. As a user-space application with access to the IPI capability, we can call this function by invoking the capability `IPICmd_Sens_Start`.

5.1.1 Create KCB

The kernel control block is obtained by retyping an aligned RAM cap of size at least `OBJSIZE_KCB` to `ObjType_KernelControlBlock`. For the core data, we later need the physical address of the freshly created KCB, but we cannot simply apply `frame_identify` to the struct capability of a KCB in user space, so we use the provided function `invoke_kcb_identify` for that.

5.1.2 Load Boot & CPU driver

Finding the ELF file and mapping the corresponding frame capability is done the same way as for spawning user-level processes to obtain a valid pointer to the corresponding ELF image in the current address space. Afterwards we allocate memory to actually load the ELF binary using the previous obtained pointer to the ELF image. The provided support function `load_elf_binary` also takes the virtual address of the entry point as argument which is used to determine the physical address of this symbol. The virtual address of the entry point is included in the ELF64 symbol table entry which can be found by name (`arch_init` and `boot_entry_psci`). The obtained physical address for the boot driver is later used in the spawn invocation and is start point where the booted core will start executing. The boot driver will then setup the environment (e.g. default kernel page table, kernel virtual address space) and jump into the virtual address space to the CPU driver entry point. While the boot driver runs with a 1:1 virtual to physical address mapping, the CPU driver is expected to be loaded at offset `ARMV8_KERNEL_OFFSET` which is added to the previously obtained entry point and also used as `load_offset` argument for ELF relocation while the boot driver just used offset 0.

5.1.3 Coredata

To provide the CPU driver with all the previously prepared parameters, we use the structure `armv8_core_data` with the fields set accordingly. The context argument of the spawn call must be the physical address of this structure, which is loaded into register $x0$. The following fields in coredata are set in user space:

- CPU driver stack and its limit (stack grows downwards)
- entry point of the CPU driver (virtual address)
- CPU driver command line arguments that are simply set to zero when not in use
- memory region to be used for the new CPU driver's allocation and the space for the `init` process

- URPC frame which is mapped by the new CPU driver at a predefined address
- monitor binary in which the CPU driver will look for an ELF image (`init` process)
- physical address of the created Kernel Control Block
- logical cores of the source (current core) and destination (new booted core)

5.1.4 Clean the cache

Before we invoke the inter-process interrupt capability, we need to make sure that everything we just wrote is visible to the new core. To do this, we first use a full system barrier and then the supplied cache maintenance function `arm64_dcache_wbinv_range` to invalidate and clean the coredata.

5.2 Multicore Memory Management

The way we manage the available memory between cores is quite simple, each core has a memory manager service running on `init`, the same that was before just running on `init` on core 0. Before we now boot up a second core, we request half of the available memory from our current memory manager on core 0 (which received all the initial ram capabilities). In order to construct all the `mem_region` for the `bootinfo` of the second core, we have to copy all the `RegionType_Module` ones and skip the `RegionType_Empty` which were previously used for core 0. The `mem_region` corresponding to the requested half of the available memory is then added as `RegionType_Empty` to the new `bootinfo` which is used in the same way as on core 0 to initialize the memory manager but now located on the new core. We also add a `mem_region` for the `cap_mmstrings` to the `bootinfo` structure to have all address ranges which are used for the spawn code to work.

So on core 0 we basically just copy the the relevant `mem_region` to the new `bootinfo` for the second core. However, in order to actually use the provided address ranges on core 1, the appropriate capability is needed. To achieve this, when receiving the `bootinfo` on the second core, we used `ram_forge` for the `RegionType_Empty` and `frame_forge` for the `RegionType_Module` to obtain a valid capability for the described address ranges. This process will change as we revisit capabilities later in the individual milestones described in Chapter 12. It is worth noting here that the ability to forge a capability is another operation that only privileged user-space processes like our `init` domains can invoke.

5.3 Inter-Core Communication

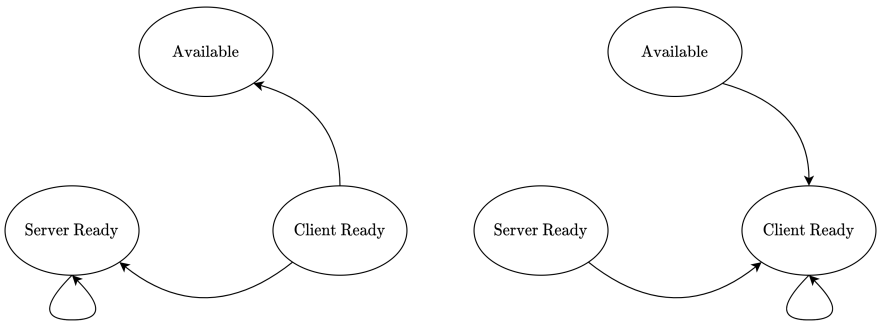
After booting the second core the next goal was to enable communication between cores using shared memory. Recall the URPC frame that was included in the core data when the second core was booted and that was allocated and mapped on the bootstrap core. As mentioned earlier, the new CPU driver also mapped this frame to a predefined address `MON_URPC_VBASE`, which is already the basis for our initial simple user-level communication channel. The protocol for reading and writing the shared memory is a simple ring buffer-like approach, where the ring buffer contains only one slot that represents the entire URPC frame for now.

At the beginning of the frame we use a status flag to indicate whether the frame is unused and ready to write or ready to read. Figure 5.1 shows the state machines corresponding to the master respectively slave. Originally, we implemented only simple client-server communication: the server waits until the client signals that it has written data for the server to the shared frame. It then processes the request. If there is a response the server sets the flag to "Server Ready" once it has been written to the URPC frame. Otherwise, it sets the flag to "Available". The client can start to make a request whenever the frame is "Available", i.e., currently not used for transfer. After writing the request and setting the flag to "Client Ready", it can wait for a response ("Server Ready" flag) or just overwrite the frame with the next request after the server is done processing the current request.

For example, the client sends a spawn request to the new core and then waits until it can read the success response from the shared frame. The server constantly checks the status flag in the shared memory, processes the spawn request once the client is finished writing, and writes the spawn reply to the frame before signalling that it is done processing this request. Sections 6.1 and 6.2 describe how we later removed this busy looping on the status flag by integrating ump into waitsets.

In addition to the 1-byte status flag, the URPC frame metadata also contains a 1-byte type field and an 8-byte length field. There are only two types `AOS_URPC_TYPE_RPC` and `AOS_URPC_TYPE_BOOTINFO`, described in Section 5.3.1 and Section 5.3.2. The length field specifies how much data is transferred in the frame. This gives an overhead of 10 bytes, which we considered acceptable as the entire frame can transmit 4096 bytes in total.

We completely revised this design in the next milestone, as described in Section 6.2.



(a) server (b) client

Figure 5.1: Basic URPC state machines.

5.3.1 Encapsulate RPC Messages

We decided to encapsulate RPC messages: the URPC layer considers RPC messages as opaque byte objects and transmits the entire object over the frame. This means the layer does not understand nor parses RPC messages, it just transmits them over the URPC frame.

This design has the big advantage of being very simple and quickly enables the transmission of small RPC messages that do not transmit capabilities. However, we are not able to transmit capabilities or large RPC messages. The reason for the latter is that messages which do not fit into a single LMP message are transmitted over a shared page: they are split into one or more 4096-byte chunks. For each chunk, an RPC message with metadata is sent over LMP, describing how many bytes can be read from the shared frame. To support sending such large messages over URPC, the URPC layer would need to recognize and parse those RPC messages, read the data from the shared frame, and transmit it over the URPC frame. Additionally, it would need to communicate to the receiving core, how much data is transmitted. Instead of extending URPC in this way, we decided to use a unified message passing layer described in Section 6.1.2.

5.3.2 Transmitting the Core Bootinfo

Since the bootinfo is larger than 32 bytes, we introduced a special message type for transmitting it. The `init` process directly uses the URPC layer (without going through RPC or LMP) and writes the entire bootinfo into the page. It sets the URPC frame's length and type for this bootinfo struct (which fits into a single page). In the redesigned message passing described in Section 6.1.2, we removed the size limitation from the URPC channel, which enabled us to integrate sending the bootinfo into normal RPC messages.

5.3.3 Barriers

Due to the weak memory model, the writes and reads from another core are not guaranteed to happen in program order. To ensure correctness, i.e., that we only read valid data from the URPC frame, we need the following memory barriers:

- **Before changing the status flag:** there are two cases: first, after writing to the URPC frame, we need a barrier that makes sure the the data is written to memory before we write the status flag. Otherwise, the receiver could start reading before all data was written to the frame, and thus read inconsistent data. Second, for making the frame available after reading, we need a barrier to make sure all data was read before we change the status.
- **After detecting status flag change:** after the code that busy loops on the status flag, we need to add a barrier to prevent later instructions from already executing and reading from the frame, before we even read the updated status flag.

Chapter 6

Milestone 6: User-Level Message Passing

Milestone 6 consisted of a complete refactoring and major redesign of our entire message passing system. Section 6.1 first gives a high-level overview of the changes and explains the message passing (MP) layer, which we introduced between RPC and LMP/UMP. Section 6.2 discusses the evolution of the simple ICC protocol from Section 5.3 to a more sophisticated inter-core communication protocol that uses cache-to-cache transfers instead of just shared memory. Finally, Section 6.6 describes the implications of booting a second core on our monitor server, and how different processes are managed across cores.

6.1 Message Passing Design Evolution

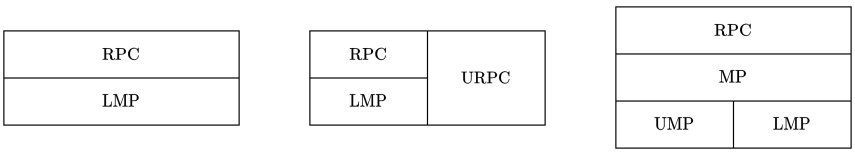
In this section, we first give an overview on the design changes compared to previous milestones. Afterwards, we describe the new message passing layer that multiplexes between LMP and UMP and makes the different channels transparent to RPC.

6.1.1 Overview

Figure 6.1 gives an overview of the evolution of our message passing design.

First, we only had a single core. As described in Section 3.1.1, we created an LMP layer that provides functions to send and receive 32 byte messages. On top of this, we added an RPC layer with different message types and support for longer messages. The latter are fragmented and assembled by the RPC layer.

Second, we added simple inter-core communication between cores (cf. Section 5.3).



(a) M3: single core (b) M5: simple URPC (c) M6: MP

Figure 6.1: Evolution of the message passing abstraction layers

For this purpose, we added a URPC layer that is used directly by applications. RPC messages for another core were encapsulated and sent directly over URPC instead of RPC and LMP. The receiving core extracted those RPC messages using the URPC abstraction and then locally sent them over RPC if necessary. There was no support for inter-core messages that are larger than the shared page.

Third, in this Milestone, we redesign the entire stack and integrate the inter-core messaging into our RPC abstraction. For this purpose, we add a new Message Passing (MP) layer, that selects the appropriate channel (UMP or LMP) depending on the message context. The LMP and UMP layers both provide functions for sending fixed-sized small messages. MP performs the fragmentation and assembly to support larger message sending over both channels. We describe the MP layer and our design choices in Section 6.1.2 in more detail. The new UMP layer, which is a complete redesign of the former URPC inter-core communication, is discussed in Section 6.2.

6.1.2 Message Passing Layer

The MP layer introduces a message channel struct `mp_chan` shown in Listing 3. RPC channels now operate over MP channels, which can either be of type `MP_CHAN_LMP` or `MP_CHAN_UMP`. We pushed the metadata tracking for sending large messages (which was described in Section 3.1.3) from the RPC layer down to the MP layer. This provides a higher-level abstraction of arbitrary size message sending to the RPC layer. Furthermore, `mp_chan` tracks a frame used for large message transfer of LMP messages. Moreover, the MP layer allows the caller to register a callback function (the signature of which is shown in Listing 4), which is called for received and pre-processed messages. MP implements a wrapper callback function for every channel, which is registered on the underlying LMP/UMP channel (both are integrated into waitsets, cf. Section 6.2). This wrapper reassembles messages received on the underlying channel, and then calls the registered callback function with a unified RPC message. Therefore, the RPC layer is independent of the underlying communication channel, i.e., whether core-local or inter-core

communication is needed to transmit a message.

```
struct mp_chan {
    union mp_chan_union u;
    enum mp_chan_type type;

    // Large message sending metadata
    bool large_in_progress;
    size_t size_total;
    size_t size_remain;

    // Thread-safety of MP
    struct thread_mutex mutex;

    // Frame used for large message transfer of LMP
    struct capref transfer_buf_cap;
    void *transfer_buf;
    size_t transfer_buf_size;

    // Callback function called on received messages
    mp_handler_t callback_func;
    void *callback_data;
};
```

Listing 3: Layout of a message passing channel struct.

```
enum mp_chan_type {
    MP_CHAN_LMP,
    MP_CHAN_UMP
};

union mp_chan_union {
    struct mp_lmp_chan *lmp;
    struct mp_ump_chan *ump;
};

typedef errval_t (*mp_handler_t)(struct mp_chan *chan,
                                 struct aos_rpc_msg *msg,
                                 struct capref cap,
                                 void *data);
```

Listing 4: Supporting data structures for the MP channels: channel type, channel pointer union, and callback function signature.

LMP/UMP Branching Initially, one motivation for the introduction of the MP layer was to share message fragmenting and assembling code, which is needed for both LMP and UMP. However, this turned out to conflict with the performance-related goal of minimizing the number of data copies. There are two underlying problems: for sending, the UMP layer requires us to send a full cache line to the communicating party. We want to avoid leaking memory in the case where the transferred message is smaller than a cache line. To achieve this, we copy smaller UMP messages into a buffer of cache line size initialized to all zeros. For receiving, we also have an issue: our basic RPC message structs, initially designed for LMP transfers, are 32 bytes. Anything that is larger needs to be allocated on the heap. Therefore, when we receive a full cache line over UMP, we first need to store it in a buffer that is large enough and inspect the type of the message. For large messages, we need to receive and assemble multiple messages. Finally, we copy the received data to a malloced RPC message of appropriate size. In summary, UMP messages require memory copies that LMP does not need. This difference between LMP and UMP is mainly due to fundamental design decisions on the RPC message struct layout. We created it with LMP in mind and this turned out to be sub-optimal for UMP. Although we refactored a lot of the message passing, we did not change the RPC messages, since this would have essentially meant re-implementing M3. As a consequence of avoiding unnecessary copies for LMP, there are quite a few separate code paths for LMP and UMP in our MP layer. The downside of this approach is that it increases the complexity of the MP code and makes it harder to maintain. In retrospective, the implementation would have been cleaner if we pushed down the fragmenting/assembling to the LMP/UMP layers, i.e., completely separated those code paths. We learned that early decisions, such as the RPC message design, have very long-range impact on later design options. As the AOS book hinted on the later need for UMP messages, we tried to plan ahead and envisioned something like an MP layer for it. However, that did not save us from completely refactoring the entire message passing multiple times. We found it to be very hard to already incorporate future message channel types into a design, without knowing their specific use cases or requirements.

Transparent Channels Another goal of the redesign was to make message channels transparent to the RPC layer and all applications using the RPC abstraction. As mentioned before, the `mp_chan` contains a type variable specifying whether it is an LMP or UMP channel. In the first MP layer design, we added the PID to the RPC channel initialization. An application that wants to connect to another application provides the PID of the latter. Internally, the RPC layer passes this PID to the MP layer, which decides whether intra-core or inter-core communication is needed. Since we route all inter-core messages over `init`, any application that is not `init` gets an

LMP channel to `init` on its core, even when the receiver's PID is on another core. The PID is stored together with the channel, and `init` knows from the context or an explicit PID field in a message, that it needs to forward the message to the other core's `init` process. When `init` itself sets up an RPC channel for a PID on another core, the MP layer internally deals with setting up a shared page (used underneath for the UMP circular buffer). We store a list of RPC channels in the core state, together with the PID for which they were set up. For message forwarding (e.g., a process' message to another core), `init` looks up the RPC channel registered for the destination's PID and sends the message over it. If no such RPC channel is found, `init` throws an error. Once the channel is created, the MP layer will use the API provided by LMP respectively UMP, depending on the channel's type.

Advantages of this design are:

- Reduced application complexity; applications do not need to handle the channel setup and can just use a simple RPC interface.
- Less code reuse; the channel setup code does not need to be replicated for each channel that is initialized.

However, as we continued our project, we faced the issue that this design reduces the application's flexibility. Since the channel setup is handled completely transparently, the application can no longer influence what channel is set up and how. The MP layer makes simplifying assumptions for the setup. For example, it assumes that only core 0 sets up UMP channels to other cores. Later, when we added duplex LMP and UMP connections, we had to break this strong abstraction layer and again expose some of the channel setup to the RPC layer. Those changes are described in Section 6.3.

6.2 URPC Message Passing

As already mentioned in Section 5.3, to pass messages to a different core, we exclusively use a shared buffer. For this milestone, we integrated our simplistic cross-core message passing into our message passing stack.

The new UMP library provides the similar API as the LMP library: Blocking send and receive functions as well as the means to register a callback for incoming messages in a waitset. In contrast to the LMP library, we can send 56 bytes at a time over UMP.

The message data itself is sent through a ring buffer in the shared frame. Every element of the ring is 64 bytes, the size of a single cache line and is also 64-byte aligned. Listing 5 shows the C-struct used for each line in the ring.

The first 56 bytes (7 data words) are used for message data and the last word stores protocol flags. Currently, only one bit is stored in the protocol flags

```
struct mp_ump_cacheline {
    volatile uint64_t data[56];
    /**
     * Last word stores protocol flags.
     *
     * Setting all bits to 0, resets the line
     * (makes it ready for writing)
     */
    union {
        uint64_t raw;
        struct {
            uint64_t unused : 63;
            /**
             * 0 if this line can be written,
             * 1 if it contains data to be read
             */
            uint64_t ready : 1;
        } d;
    } flags;
};
```

Listing 5: Structure of a single data line in a UMP channel.

and it indicates whether the cache line contains data for reading (set to 1) or the line is ready for writing.

This buffer is the only way two processes on different cores can communicate, there is no kernel that spans multiple cores that could provide LMP-like events. Because of that, the receiver has to poll the `ready` bit of the next line it wants to read until the sender sets it to zero.

This exchange requires careful placements of memory barriers to ensure both sides observe relevant changes to the buffer in the correct order. Without any barriers and because of the weak memory model, it is possible for the receiver to read the buffer contents before the sender actually writes to it and once the `ready` bit is set, those old, invalid, contents are used.

To solve this, we place a memory barrier directly after we finished polling the `ready` bit and in between reading/writing the line data and changing the read bit. Figure 6.2a and Figure 6.2b show the correct barrier placement in pseudocode.

Each channel contains two ring buffers. The first is used for writing by one side and reading by the other and the second is used the other way around.

The reason each line is exactly 64 bytes and 64-byte aligned is because this

```
// While not ready for writing
while(ready bit set);
BARRIER;
write_data();
BARRIER;
// Mark ready for reading
set ready bit
move to next line
```

(a) Correctly writing a line to the shared buffer with memory barriers.

```
// While not ready for reading
while(ready bit not set);
BARRIER;
read_data();
BARRIER;
// Mark ready for writing
unset ready bit
move to next line
```

(b) Correctly reading a line to the shared buffer with memory barriers.

Figure 6.2: Pseudocode for reading and writing a line in a UMP ring buffer.

way each line fills up exactly one cache line and we can use the CPU's cache coherency protocol to exchange data between cores without going through main memory and thus massively improving latency. Because the processor uses a MOESI-like protocol for cache coherence, it can send the state of dirty cache lines to another core without touching memory, the exact process is described in detail in the AOS book [1, Section 8.5].

The code for receiving a message over the channel in Figure 6.2b is blocking, if we want to be notified of new messages, we can register the UMP channel as a polled channel in a waitset and supply a callback. All the logic behind when to poll and how to notify users is done by the waitset code (just as it is for LMP channels), we only need to provide a way to poll our channel. This is as easy as checking the `ready` bit of the current line in the receive ring.

6.3 Duplex RPC Channels

Up until now, all data transmitted over an RPC channel would travel over the same underlying LMP or UMP channel. As long as processes only used RPC to send requests to `init` and never tried to act as a server and asynchronously receive requests on that channel, this works.

As soon as a process expects to both send and receive requests on the same channel, we may run into issues. Imagine a process sending a request to `init` (this happens over LMP, but the same also applies when two `init` processes communicate over UMP), after sending the request, it waits for a response on the channel. If at the same time `init` sends a request to the same process, the process will receive the request in place of the response it expects.

This is clearly a problem because we don't receive the right response to a request, but in the case of LMP this could lead to a buffer overflow because `init` sends two messages on the channel: the request and the response to the

first request of the process.

At the time we came up with two possible solutions to this problem:

1. Send messages over LMP or UMP as packets with sequence numbers and unique IDs per transfer. The receiver would then deserialize the packets into full RPC messages and deliver them to the right place.
2. Have two dedicated LMP or UMP channels per RPC channel. One is for sending requests and receiving responses (the request channel) and the other is for receiving requests and sending responses (the listener channel).

The first option would have been a clean solution and also a nice abstraction, making the message passing stack closer to a networking stack. It would however have required another complete overhaul of the messaging stack and we did not have the time to do that.

In the end we opted for number two because it was easier to implement and did not add much complexity to the existing stack.

The main complexity this change added was in the initialization. For UMP, this was quite straightforward, we split the URPC frame in half and initialized both channels with half of the frame. Setting up LMP channels became a lot more involved. Instead of the two-step handshake where the process sends its endpoint cap and receives a frame cap for the shared frame, we now have a six-step handshake where we also exchange endpoint capabilities and frame caps for the second channel. Later, the handshake was even extended to eight steps (see Section 6.5)

With this change, any process could concurrently act as a server by registering a callback for incoming messages and send out requests.

6.4 Deadlocks in Single-Thread Message Handlers

One of the biggest issues we ran into when stress testing our messaging stack, was that under certain circumstances the message handlers in `init` would deadlock on both cores. The callback functions for new messages would run in the same thread as the one that called `event_dispatch`. This means that as long as we are in the message handler, we do not go into the next iteration of the event loop and can thus not receive any new messages. The deadlock occurred once the `init` processes on cores 0 and 1 were both in the message handler and tried to send a message to the other init process. In that case, both processes waited for the other side to receive a message while neither side could receive messages, leading to a deadlock.

In practice this could happen when running processes on both cores that try

to spawn processes on the other core. This placed a quite serious limitation on our design: As long as we are in the callback of a channel, we must not send or receive RPC messages.

Our first attempt to solve this was to make the message handler very short by just making it add the received message onto a queue and process that queue in a separate thread. With this approach, in the same deadlock situation as above, the messages did arrive on the other side and were added to the queue, but because the message processing was still sequential, we still had a deadlock because the request would never be processed by the other side.

We ended up solving this by spawning a detached thread for each incoming message. This way, the only thing blocked while processing a message was the listener channel on which the message arrived (because we need do not re-register until we send a response back).

A cleaner solution would have been a thread pool with one thread per RPC channel and on every new message we would wake up the corresponding thread to process it. However, due to time reasons, implementing a thread pool was not feasible.

6.5 Allocating Memory while Receiving a Response

Whenever a non-init domain receives a reply for a request on the request channel, we may need to allocate memory for the received message. If this allocation triggers morecore and requires us to request more ram caps, we have to send a ram request and receive a reply on the request channel in the middle of receiving another reply. Surprisingly, this does not give us any issues as long as the received message can be sent in a single transmission ($<$ 4KB), because the message on the LMP channel has already been read and the ram reply never writes to the shared frame.

But as soon as a larger reply is sent or the slot allocator runs out of slots when refilling the receive slot after receiving the ramcap, this the wrong replies to be received.

One partial solution would be to pre-allocate a buffer where we can receive large messages. However, this is impossible as the size of the reply message is not known beforehand and could have an arbitrary size. We got around this by creating a dedicated one-way ramcap LMP channel (only a requester channel on the processes and only a listener channel in init) for each non-init process as part of its RPC channel to init. All ram allocations would go through this channel, solving the issue of having to allocate memory when receiving large replies.

This required us to extended the LMP handshake from six to eight steps

because the process and init had to exchange endpoint capabilities.

The last thing left to figure out is how to deal with the slot allocator running out of slots after receiving a ramcap. Once the slot allocator has no more slots left to use as the receive slot, it is impossible for the process to receive more ramcaps. So, we need to preemptively make sure this never happens. By default, the LMP layer always has a full receive slot and refills it automatically after receiving a capability. For requesting ramcaps, this does not work as nothing in the entire request/receive operation should ever have to allocate memory. For that reason we turned of auto-refilling of slots in the ramcap channel and before any ramcaps are requested, we make sure that we have enough slots to receive it. We added an extra backup slot into the channel so that we had always at least one available slot to use after receiving a ramcap.

Before requesting any ramcaps, we now always make sure that both slots are full. The slot refilling itself can trigger another ram request, but this part is reentrant; the channel notices when it recursively tries to refill the slots and just skips the refill part (we have at least one slot available at this point and can receive a ramcap). Only after both slots are refilled, do we continue on to request the actual ramcap.

6.6 Monitor Server Extension

The ability to boot the second core also has some impact on our current monitor that manages the running processes. We have designed our system so that for each `init` domain, each core has a corresponding monitor that manages the processes running on that core. This also means that we now have to properly handle the core argument to the process spawn RPC to enable spawning processes on a specified core. The core idea is pretty simple, we just insert the core for spawning or the PID to retrieve a process name into the message sent by the client-side RPC, which is then received by the monitor handlers running on the same core. The actual work is then done in the monitor, where we decide whether the monitor itself can handle the call or whether the request must be forwarded to the monitor running in the other `init` domain on the other core. To explain this in more detail, we first need to explain some changes.

6.6.1 Refactoring the Bookkeeping

Think of the performance dependency of our chained list structure. To avoid this, the monitor now maintains an array of pointers to `monitor_node` structures (Listing 6), where each structure corresponds to a running process on the same core where the monitor is running. The process ID is used as array index to to make the name lookup as efficient as possible. But having

a monitor for each `init` domain brings up the question how we assign PIDs which are unique across cores.

```
struct monitor_node {
    enum processtype type;
    char *binary_name;
    domainid_t pid;
    struct aos_rpc *rpc;
    struct spawninfo *si;
};
```

Listing 6: Layout of a monitor node corresponding to running process on the given core.

We use a union `aos_pid_t` to split the actual `domainid_t` into a `local_pid`, `core_id` and a `unused` part. Additionally each monitor keeps track of a local `uint16_t` counter for the next PID which is then used as the `local_pid` part which also corresponds to the array index described above. Together with the `core_id`, we have unique process IDs across the cores that we can use to quickly find out which core a particular PID corresponds to. By convention we use the `local_pid` 0 for the `init` domain on each core and the maximum of the local part is now `UINT16_MAX`.

```
struct PID {
    uint16_t local_pid; ///< range 0-65535
    uint8_t core_id;    ///< byte for the actual core ID
    uint8_t unused;     ///< unused
};

typedef union monitor_PID {
    struct PID sPID;
    domainid_t dPID;
} aos_pid_t;
```

Listing 7: The process ID is split into a local and a core ID, where the lower two bytes are used for the local part in order to quickly distinguish which core a PID corresponds to, since our system uses little-endian.

6.6.2 Server-Side RPC

The messages sent by the client-side RPC functions now contain the core ID for the spawning and receiving all process IDs. We have also added an additional RPC function `aos_rpc_process_get_pids` that takes an additional argument for the core you want to get all PIDs from. By passing -1 you receive the

list of PIDs from all running cores. The server-side RPC functions within the monitor must now distinguish whether a particular request can be resolved locally, must be forwarded to another monitor running on another core (e.g. spawning on another core) or collects information from all running monitors (e.g. PIDs from all cores). As you can now see, using the `domainid_t` in a way that also decodes the core ID in it has some advantages. When requesting the name of a given PID, the monitor can directly recognise which core the PID corresponds to and forward the request. To retrieve the PID list, we have either specified a core or want to retrieve the PIDs from all cores. To do this, we either query the local PID list or forward the query to the `init` domain running on a specified core, and in the case of all cores, we combine the received lists at the end.

6.6.3 Performance implication

Replacing the linked list with an array of pointers indexed by the local part of the process ID allows most operations to be performed regardless of the number of processes actually managed by the monitor. This is crucial for spawning and translating a PID to its binary name. We still have some dependency when we assign a new process ID to a process because we have to check if the `next_pid` managed by the monitor is still in use and if so, we increment it and check again. This process is repeated for a new process until we have tried each local PID once, then we return a `SPAWN_ERR_OUT_OF_PIDS`.

This design has a major drawback when retrieving all PIDs, as we actually have to check the array of `UINT16_MAX` pointers in it and return any PID where the pointer to a `monitor_node` structure is not `NULL`. But every other operation is performance-wise independent of the number of processes actually managed at runtime, which is much more important from our point of view. Especially, since we expect the operation of collecting all PIDs to be fairly rare.

Part III

Individual Milestones

Chapter 7

Milestone 7: Individual projects: building on the core

Chapter 8

Shell

In this chapter, we explain the design of the command line. First, Section 8.1 describes how we set up device drivers, and specifically the UART driver, in user space. Moreover, this section describes console input and output. We describe the shell's core functionality – including line parsing, control characters, and built-in commands – in Section 8.2.

8.1 User-Space Console I/O

This section describes how we set up the UART driver and implement I/O over the serial console. Next, we look at the setup of interrupt forwarding for reading characters from UART efficiently. Furthermore, we discuss how the serial console is multiplexed between different processes on a line-by-line basis.

8.1.1 User Space Device Drivers

The first step for using device drivers from user space is to map the device registers. For this purpose, we first create a frame of the correct size and offset for our device. We do this by retyping the special `DevFrame` capability stored in `init`'s `TASKCN_SLOT_DEV`, which refers to a large memory region covering most registers, to smaller `ObjType_DevFrame` caps. For UART3, the base address is `0x5A090000` and the `DevFrame` region starts at `0x50000000`. Therefore, we need an offset of `0xA090000` for the capability retype operation, such that we get a frame for UART3. Afterwards, we need to map this frame uncached to the address space of the process using the UART driver. In our design, the terminal server runs on `init`, which means it was the the natural choice to simply map this frame to `init`'s address space. Other individual projects use device frames in different domains and pass them when spawning the domain.

8.1.2 Interrupt-Based Character Reading

To avoid busy-looping on the UART channel and blocking `init`, we register an interrupt handler and tell the UART to generate receive interrupts when new characters can be read. For this purpose, we have to set up generic device interrupt distribution in user space. We are provided with a GIC distributor driver, which we can initialize as described in Section 8.1.1. Afterwards, we configure the CPU driver to deliver interrupts to our domain by using the provided library functions. They provide functionality to get an IRQ destination cap and register an endpoint for the UART3 interrupt vector in this `IRQDest` capability. Afterwards, we enable the forwarding to our CPU and with high priority in the GIC distributor. Finally, we use an instruction barrier to make sure all previous operations are done and the interrupt handlers are set up correctly, before we enable them in the LPUART. This makes sure that the CPU driver does not receive an interrupt which it cannot handle.

In the interrupt handler that we register, we implement a short polling sequence. This is necessary because we find that not every single character generates an interrupt. Thus, we need to read for as long as there are characters to not miss any input. After ten `LPUART_ERR_NO_DATA` errors, we assume that there is no immediate new character and wait for the next interrupt. Note that it does not matter when we already consumed a character that also generated a new interrupt: the next run of the interrupt handler might not have characters to read, but it is robust against those `LPUART_ERR_NO_DATA` errors anyway.

8.1.3 Multiplexing

We first give a high-level overview. We multiplex the single serial console port between different domains on a line-by-line basis: whenever a process requests a character, it claims the entire line. Any subsequent input on the same line is received asynchronously and buffered in line buffers described below. Later character requests from the same process are served from the line buffer. Other processes wait until the line is finished before one of them can claim the next input line. Output is multiplexed in a similar fashion: once a process outputs a character, the entire line is claimed for that process. This process is afterwards the only one that can continue to output characters, until it outputs an end-of-line (newline, carriage return, or end-of-file) character. Other processes wait until the serial port is unlocked, before writing characters to it.

We now discuss the technical details and choices of this implementation. Listing 8 shows the internal representation of the serial driver. The `mutex` is used to serialize accesses to the single serial port. Our terminal server runs

on `init` and receives AOS RPC messages for receiving or outputting single characters respectively strings to the serial console. Since every process has its own MP channel to `init`, we distinguish different processes based on their channel. To claim a line of input or output, a process tries to lock the `mutex`. Once it has the lock, it sets itself as the current `owner` of the serial port. This lock is only released when the process finished receiving or outputting a line. The current owner of the serial port can always use it to receive input or send output. All other domains need to wait until the mutex is unlocked.

```
struct serial_driver {
    struct lpuart_s* lpuart;

    struct thread_mutex mutex; // serialize access
    volatile struct mp_chan* owner; // Current owner of the
    ↪  port

    // Line buffer and metadata management
    volatile uint32_t *get_char_idx;
    uint32_t get_char_buf_len;
    char *get_char_buf;
    volatile bool *is_done;

    bool locked; // True when the current line is locked
};
```

Listing 8: Internal object representing the driver for our single serial port.

Since a single RPC message may only request part of a line – often only a single character – we need to buffer received lines asynchronously. The reason for this is that our interrupt handler is called whenever a new character arrives. We need to store this character and cannot wait for the next AOS RPC request of the owner domain. The reason is that the LPUART device does not generate new interrupts before the current characters are consumed. Therefore, we allocate a line buffer (shown in Listing 9) for every channel request. This data structure is a linked list of line buffers for MP channels. Each line buffer stores a buffer `buf` of the maximal line length `buf_len` and some metadata. The `init` process can check for `getchar` messages, whether they already have a line buffer. If they do, then it reads the next character at index `read_idx` from this buffer. The next character is written to the index `write_idx` in `buf`. If the line is finished, `is_done` is set to `true`. Until that point, `init` can wait until `read_idx` is smaller than `write_idx` and then return that new character and increment `read_idx`. We need this data structure for every domain that has an outstanding request, because there could be multiple lines that were received for different domains, but the

domains did not yet consume all characters of those lines. Thus, `init` needs to continue to buffer them, until the last character was requested. Meanwhile, other domains can receive data from the next input lines.

```
struct serial_line_buffer {
    struct serial_line_buffer *next;
    struct mp_chan *chan;

    // Buffer and metadata for this line
    char *buf;
    uint32_t buf_len;
    volatile uint32_t read_idx;
    volatile uint32_t write_idx;
    bool is_done;
};
```

Listing 9: Line buffer data structure to asynchronously receive serial console input.

One disadvantage of this rather strict line multiplexing is that any process can prevent all others from using the serial console. Once a process claimed the current line, no other process can print or get a character until the current owner prints a newline. This can obviously block the entire system. Moreover, in the case of the shell, this caused some counter-intuitive behaviour for us. When you start a process detached from the shell, the shell can claim the current line of the serial console for printing the shell prompt and waiting for the next command, before the spawned process can print. As a consequence, `printf`s of the latter will only appear after the user hits enter on the command line. The reason for this is that the shell then releases the lock on the serial console and the spawned process can grab it for its output. To improve this, we slightly relaxed the line multiplexing: RPC calls for getting/putting characters now support a flag that specifies if a process wants to lock the line or not. We added the `locked` boolean to the `struct serial_driver` (in Listing 8), which is set to `true` as soon as at least one RPC message specified to lock the line. Most messages always lock their lines (including normal `printf` calls), but the shell no longer locks it when printing the shell prompt. Before the user enters any characters on the prompt, other detached processes are allowed to print and/or claim the current line for input. But as soon as the user entered a character for a new command, the shell claims the line and no other process can print or get characters interleaving with the currently entered command.

The command `demo` (cf. Appendix A) demonstrates this line multiplexing for processes competing for command line output (`-o` flag) as well as input (`-i`

flag). The input test spawns two processes that both wait for a second and then request input from the console. Listing 10 show cases line multiplexing: first, only the process with PID 5 asks for input. After we enter `abc` and hit enter, the second process, which waited up to that point, asks for input. We type `123` and hit enter. Now the processes repeat the entered input, showing that they really only received the characters in their input line. A similar test to show case output multiplexing is implemented by `demo -o`. In that case, two processes print their first argument a thousand times on the same line. We spawn them with the numbers `0` and `1`. Without line multiplexing, the resulting output would have interleaved zeros and ones. Instead, our output prints two properly separate groups groups of `0` and `1`.

```
$ demo -i
PID 4 starts two processes competing for console input
Process with PID 5 started
Process with PID 6 started
PID 5, enter input: abc
PID 6, enter input: 123
PID 5, read: abc
PID 6, read: 123
```

Listing 10: Input test case implemented by `demo -i`.

8.1.4 RPC Message Integration

In general, there are RPC messages for getting and putting individual characters. The shell uses those to fetch single characters, filter control characters (cf. Section 8.2.3), and only put printable characters on the serial console for the user to see what he is typing. This even works on different cores, since RPC messages are transparently forwarded to `init` on core 0, which runs the user-space serial console driver. In other words, it is possible to spawn another shell on a different core, using the sequence of commands shown in Listing 11. For more information on the commands, see the user guide in Appendix A.

```
$ info -c -p
Core id: 0
PID: 4
$ oncore -a -c 1 spawnShell
Spawned process 'spawnShell' with PID 65537 on core 1
Starting a shell...
$ info -c -p
Core id: 1
PID: 65537
$ exit

End of shell
$ info -c -p
Core id: 0
PID: 4
```

Listing 11: Sequence of commands to spawn a shell on core 1. The `oncore` command must be used with the attach flag, so that we switch the current shell to the newly spawned one.

However, not all applications need to operate on single characters. We want to avoid the overhead of sending two RPC messages for getting a character (the `getc` and the reply with the character), and two additional ones for echoing it in the console (the `putc` and an `ack`). Therefore, we added the functionality to print an entire string as well as to get a complete line. Both only make one invocation of the RPC abstraction. If the transmitted data (the string to print or the line read from the console) is too large, this still sends multiple messages (the size of which depends on the underlying channel), but we still send significantly fewer messages than we sent before. We also need far less acknowledges, since we only acknowledge each group of characters instead of the single ones individually.

For `init` on core 0, we had to implement a loopback RPC channel to enable it to print over our user-space driver. The challenge was that after replacing the `libc` printing stubs, strings arrive in the registered function in `init`. Although our console server runs in `init` too, it is called from the RPC message handlers, which is spawned in separate threads, so that it can block for inputs. Thus, in order to integrate this reasonably with our existing message handling, we decided to create a loopback RPC channel for `init` and let it send the RPC messages for getting/putting characters to itself. Afterwards, a message handler thread of `init` is created and calls the serial driver as for all other domains. This is a special case for `init` on core 0, because the other cores are already set up to forward terminal server requests to core 0.

8.2 Shell Core Functionality

This section first gives an overview over the general design of the shell. Next, we explain the command line parsing state machines. Furthermore, we discuss the supported control characters on the command line. Finally, we give a brief overview on the supported built-in commands.

8.2.1 General Design

At a very high-level, in our design a shell is an object storing context and commands. Spawning a shell involves instantiating such a `shell` object (see Listing 12) and then receiving input lines, parsing them to commands, and either execute the command or provide diagnostic output.

The `shell` stores context information (similar to environment variables). It stores the current working directory and the shell's home directory, as well as a variable that is `true` when the shell is currently running, i.e., it is parsing and executing commands. Furthermore, the shell object stores a list of the available commands and their format in `cmds`. We now describe those commands in more detail.

```
struct shell {
    struct shell_cmd *cmds;
    bool is_running;
    char *working_dir;
    char *home_dir;
};
```

Listing 12: Shell object storing execution context and commands.

Commands need to be registered with a shell object. We implemented a shell commands API that facilitates command creation and registration. Listing 13 shows the format of a command. The `shell` keeps a linked list of registered commands, which are `shell_cmd` structs of type `SHELL_CMD_TYPE_REG`.

There are two types of `shell_cmd`:

- `SHELL_CMD_TYPE_REG`: This command defines the allowed format for a command. It is required to define the command's `name`, a `help` string, and the which options the command takes. Those options are stored in a `shell_cmd_args` struct shown in Listing 14. Each command argument needs to have a short flag `short_name` (format: `-[a-zA-Z]`) or a long flag `long_name` (format: `--[a-zA-Z]+[a-zA-Z-_]*[a-zA-Z]+`) or both. Every command argument for a `SHELL_CMD_TYPE_REG` command needs to have a short `help` string. The flag `required` indicates whether this argument is required to be present or optional. A command argument

can have arguments itself, which are the strings following the option flag, which do not start with a dash. The format defines that the command has at least `argc_required`. The `argv` field stores `argc` pointers to names of the arguments in case you want to name positional arguments. Those are displayed in the help pages.

- `SHELL_CMD_TYPE_PARSED`: This command is the result of the parser. It represents an input line parsed according to our format into a command struct `shell_cmd` with argument structs `shell_cmd_args`. Every argument can have arguments itself in the `argv` vector. This type of command enforces fewer fields, e.g., there is no `help` string in parsed commands. However, commands of type `SHELL_CMD_TYPE_PARSED` are still guaranteed to have a correct format in terms of names and options, since the parser fails for invalid input lines (e.g., not terminated double quotes).

```
enum shell_cmd_type {
    SHELL_CMD_TYPE_REG,
    SHELL_CMD_TYPE_PARSED,
};

struct shell_cmd {
    enum shell_cmd_type type;
    struct shell_cmd *next;
    char *name;
    struct shell_cmd_args *argv;
    char *help;
    shell_cmd_fn cmd_fn;
};
```

Listing 13: Shell command either representing a parsed command or the registered format command format.

```
struct shell_cmd_args {
    struct shell_cmd_args *next;
    char *short_name;
    char *long_name;
    char *help;
    bool required;
    uint32_t argc_required;
    uint32_t argc;
    char **argv;
};
```

Listing 14: Shell command argument, which can have a set of strings as arguments itself.

There are two main design goals in registering commands with explicit argument formats:

1. **Enforce unified command format:** we can control commands before registering them in a shell object. Thus, the library can validate commands and ensure they follow a particular format. For example, we reserve the `-h` and `--help` flags and reject any commands that register those options. Since we force all commands to add `help` strings, we can auto-generate usage pages for commands based on the command format, including `help` strings for the command and all available arguments. Therefore, we automatically add `-h` and `--help` options that print usage information to all commands, without them needing to specify this.
2. **Facilitate command parsing:** since command formats are registered in the shell, the shell can parse an input line according to the format and verify that the invocation is valid. Every command line utility registers a callback function `cmd_fn`, which is the entry point to the implemented command. It will only be called with valid, parsed `shell_cmd_args` arguments. This simplifies the utility's code tremendously, since it does not need to handle argument parsing. For example, we allow commands to specify the required number of arguments. If those are not given, the shell already throws an error before the command is even executed. The command no longer needs to do this error checking. The shell can also provide detailed parse error diagnostics, since it knows exactly how the line deviates from the intended command format. This is centralized and done consistently for all commands. In this way, we both improve the diagnostic output of malformed commands as well as reduce the command utility developer's effort. In summary, we hope to increase the overall quality of command utilities and the CLI user experience.

Another idea which goes somewhat beyond what we implemented, is that

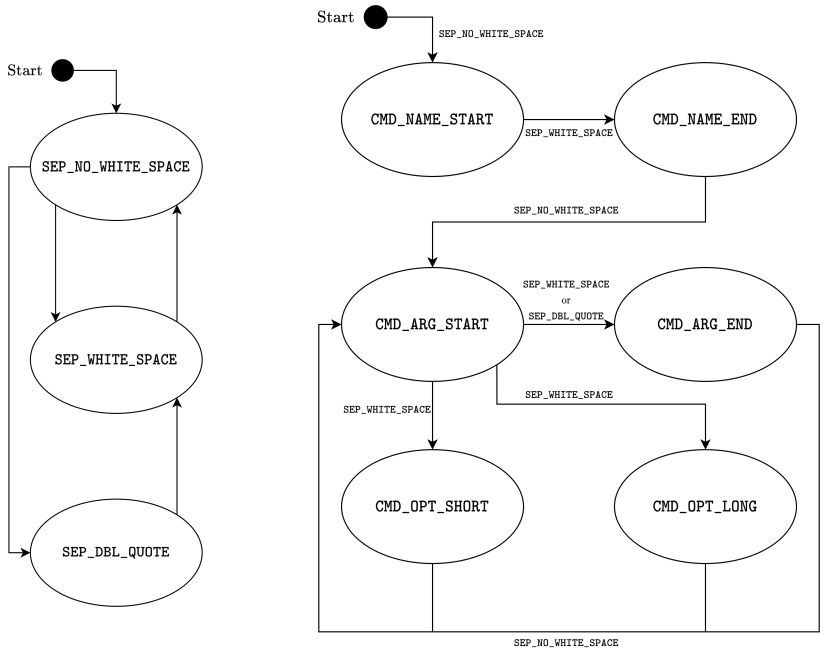
explicitly registering commands for shells provides better security than taking them from some path. This could potentially be integrated with the capability system, such that only programs with the correct capability are allowed to register more commands to a shell. In our design, each spawned shell has its own set of default commands. One can extend this list of commands at runtime, but that only modifies your currently running `shell`. You cannot place a maliciously binary somewhere on the path, that is then executed by accident by a different user[1]. You can only run commands that you explicitly register (apart from the built-in ones). This design should make it easier to define shells with less functionality or rights, since we can simply remove those commands from that user's `shell` object and do not need to work with complex permission settings. E.g., we could spawn a shell with reduced functionality for less privileged users when they log in over SSH.

8.2.2 Parsing

The parser is independent of the `shell` object. It simply takes a string as input and tries to parse it to a `shell_cmd`. If this is not successful, the parser will return an `errval_t` with an appropriate error message, which is caught and printed by the shell. If the line parses successfully, the shell tries to execute in the context of the current `shell` object. There, information on the registered commands is available and the parsed command will be validated. The purpose of separating parsing and validating is, that – in the future – different parsers/validators could be used. Moreover, the parsing or validation could also be used independently. E.g., for diagnostic output, we can validate a command and then print the error message.

The parser is implemented as the product state machine of a simple state machine `S` to keep track of the next separator (Figure 8.1a) and another state machine `C` to track which part of the command is currently parsed (Figure 8.1b). The latter annotates the state switch in `S` on the arrows between two states. To parse a command, `S` starts with state `SEP_NO_WHITE_SPACE` and `C` in state `CMD_NAME_START`. First, the separator state machine skips parsed characters until it encounters the first non white space character. This triggers the start of parsing the command's name and we change `C` to state `CMD_NAME_END`. `S` is changed to `SEP_WHITE_SPACE`, because all characters up to the next white space should be part of the command's name. On the next white space, `S` stops skipping and `C` stores the command's name and moves to argument parsing. This always starts with `C` in state `CMD_ARG_START` and `S` in state `SEP_NO_WHITE_SPACE`. On the next non white space character, we consider the first characters to decide what type of argument is parsed. If

[1] In a normal UNIX shell, consider the following scenario: say a malicious user writes a binary with the name `gti` instead of `git` to some folder on another user's path. Since people often flip letters when typing, chances are the unsuspecting user accidentally executes the malicious `gti`.



(a) separator SM `S`

(b) command SM `C`

Figure 8.1: State machines (SM) for the shell command parser.

there is a single dash, we parse a short optional argument and `C` goes to state `CMD_OPT_SHORT`. For two dashes, we parse a long optional argument and thus `C` goes to `CMD_OPT_LONG`. Otherwise, we either parse a positional argument of the command, or an argument of an optional argument. In both cases, `C` goes to the `CMD_ARG_END` state. If the first character was a double quote, the separator `S` waits for a terminating quote in state `SEP_DBL_QUOTE` and will jump over all intermediate characters and white spaces. Otherwise, `S` is in state `SEP_WHITE_SPACE` and we parse the argument until the next white space appears. Depending on the argument type, we add the argument in the appropriate place to our parsed `shell_cmd`. Afterwards, we go back to parsing arguments until we reach the end of the parsed line. If we encounter an unexpected parsing state or an invalid end state (e.g., `SEP_DBL_QUOTE` of `S` means there is a missing double quote), the parser throws an error.

The main goal of this design was to create a parser which is robust to white spaces, allows advanced options and flags, but is still understandable. We think the explicit state machine abstraction helps to avoid errors in the parsing logic and makes it easier to structure the code.

8.2.3 Control Characters

The most basic command line implementation supports no editing features: a written character cannot be deleted, the cursor cannot be moved, and new characters can only be appended to the end of the line. This makes using the shell really cumbersome. Therefore, we added support for the following control characters:

- **Backspace or delete:** delete the last character on the line (but only up to the prompt).
- **CTRL + A:** jump to the start of the line
- **CTRL + E:** jump to the end of the line
- **Arrow left:** move cursor left (up until the command prompt)
- **Arrow right:** move cursor right (up until after the last character on the line)

To make the above commands useful, we provide the option to insert characters. The easiest implementation is to overwrite existing characters. However, this has poor usability and we are very used to inserting characters. On an insertion, all characters to the right of the cursor are shifted one position to the right and the new character is placed below the cursor. This is actually not as straightforward to implement as one might think at first. We implement this as follows: first, we store the cursor's position, then we delete the line on the console from the cursor to the end (using the control character `\x1b[K`), then we shift the internally stored characters to the right by one position and insert the new character at the cursor's position. Afterwards, we print the new line to the console, starting from the new character. Next, we need to reset the cursor's position, because after printing the edited line, it is at the line's end. Finally, we need to move the cursor forward by one, since from the user's perspective, a single new character was written and he expects the cursor to be after this new character.

Some of the above commands, such as the arrow keys, do not consist of a single char. Instead, they are ANSI escape sequences of multiple characters. They start with an ASCII escape code `\x1b` followed by a right bracket `[`. The third character then signals the operation, e.g., `C` for a right arrow and `D` for a left arrow. When parsing input, we need to be careful to not include such escaped sequences in our parsed command. They are really only intended to be interpreted by the terminal.

8.2.4 Built-In Commands

In this subsection, we give the list of implemented commands, together with a one-liner of their purpose. For more information on the commands, use the

`help` command or consider the user guide in Appendix A.

The shell supports the following built-in commands:

- **cat:** output content of the file on the given path.
- **cd:** change the shell's current working directory to the given path.
- **demo:** demonstrate working user-level threads and serial channel multiplexing.
- **echo:** prints its arguments to the shell.
- **exit:** exit this shell.
- **help:** prints all available commands and help information on them.
- **info:** prints information on the system (PID, core id, thread id, supported keyboard shortcuts).
- **led:** turn LED4 on the Aster Carrier Board on/off.
- **ls:** lists the content of directories.
- **mkdir:** create a directory at the given path.
- **nslist:** list all active services.
- **nslookup:** lookup service through nameserver.
- **nsvalidate:** validate a nameservice name.
- **oncore:** run a module on the specified core.
- **ping:** sends ICMP echo requests to the given IP and print the received echo replies.
- **ps:** displays currently running processes.
- **pwd:** print current working directory.
- **rm:** remove the file at the given path.
- **rmdir:** remove the directory at the given path.
- **run:** execute shell commands stored in the given file.
- **send:** send a message to the named server.
- **sendUdp:** send an UDP package including the given data to the specified IP and port.
- **write:** Write or append string to file.

Chapter 9

Filesystem

In this chapter we talk about the design of the filesystem. In Section 9.1 we show the architectural decisions made and how those affect the usability. Section 9.2 describes the setup and functionality of our SDHC driver interface. In Section 9.3 we talk about the filesystem we chose, how we represent it internally and how we interact with the SD-card.

9.1 General Design of the Filesystem

We first give an overview of the layout we chose and the reasoning behind it. We then discuss potential downsides and how they could be fixed. The implemented filesystem itself loosely follows the `FAT32`-specification. It assumes that the SD-card is formatted in a certain way:

- The filesystem on the card is to start directly at the beginning of the card, i.e. no partitioning tables or similar should be located beforehand.
- The size of blocks which we use to access the card is fixed to 512 bytes.
- Any files on the SD-card should be in FAT32-format with 512 bytes per sector and 8 sectors per cluster.
- Any filename has to follow the FAT32- shortname specification.

9.1.1 Filesystem Layout

Our filesystem has two major parts: The `SDHC-interface` server and the `FAT32` library. The `SDHC-interface` server talks to the SD-card via the given sdhc driver by reading and writing blocks of a fixed size to/from the card. On startup, the server initializes the sdhc driver and registers itself on the nameserver. This allows the `SDHC-interface` to expose the read- and

write-block functions of the given sdhc driver, such that any other process that has the filesystem setup can write to the SD-card.

The idea here was that the `FAT32` library still does the decomposition of its files and its content to blocks itself, but does not have to worry about how exactly they are written to the card.

The `FAT32` library itself offers several functions that are called from the filehandling function of the C-library (`fopen,...`). Once it is initialized, it keeps a reference to the root folder in memory permanently, with a name-reference to each of its entries (files and folders) as its children. This partial tree gets extended each time we load a file/folder and shrunk, once we close the item again. We do this to minimize the memory footprint of the bookkeeping. The bookkeeping is needed, as we can have multiple handles to an entry at the same time, which means some operations such as delete or write can not directly be executed and have to wait.

As was required, the filesystem is "mounted" under `/sdcard/`. As we did not want to create wrapper functions that delegate the respective file functions to the correct filesystem functions, depending if we access the SD-card (`FAT32`), or `'/'` (`ramfs`), we opted to only have the `FAT32` system running. This means any file access outside of `/sdcard/` is looked at as invalid.

Currently the library writes any change to the filesystem directly back to the SD-card by utilizing the two RPC-calls that the `SDHC-interface` exposes.

We wanted to implement the main part of the filesystem as a library with the idea that you only initialize the `SDHC-inerface` once and you can then write to the filesystem anywhere you want to, without having to go through a setup everytime. This assumption turned out to be not correct, as the `libc` functions have to be set for every process anyways, which means we have to initialize the filesystem library on each process explicity anyways.

9.1.2 Improvements

As mentioned, the filesystem library has to be initialized for every process explicitly. This also means that for each process we setup the same bookkeeping again, and it does not know about any other process that currently has the filesystem open. This makes the bookkeeping useless once the filesystem runs concurrently on multiple processes and accesses the same files. It even has the downside that the stored information is now likely incorrect and takes up additional space. Looking back it would have made more sense to expose the filesystem itself as a server, such that we can have a global bookkeeping going on. Any operation, such as reading, writing etc. would then be sent to the fileserver via RPC-calls and the server would then manage access to entries there.

This would mean that we would have less overall RPC-calls while the fileserver is running due to not sending each block separately, but just having one call per operation if nothing interferes. Additionally, we would send less data around, as now we always send an entire block via RPC, even if we would only fetch a small 32bit entry from the FAT-table. This is also a direct consequence of exposing the read- and write- functions of the given driver, without doing any pre-processing of the data.

9.2 SDHC Interface

9.2.1 Setup

After spawning the nameserver in `init`, we spawn our `SDHC-interface` process. As the interface will need to spawn the SDHC-driver in its own userspace, it additionally needs access to the `TASCN_SLOT_DEV` capability that is in `init`s `TASKCN` cnode. To get access to that capability we copy it to the cspace of the `SDHC-interface` when spawning it. We only copy the capability if we spawn the `SDHC-interface`, to not expose the capability to any program spawned.

Once spawned, the `SDHC-interface` allocates and maps a memory frame of of size 4096KB. It saves the physical and virtual addresses of this frame as a global variable to mark where we want to have the SDHC-driver do its DMA to.

Next we retype the capability in the `TASKCN_SLOT_DEV` slot of the `TASKCN` capability to get the `ObjType_DevFrame` of our external SD-card. We then map that range into our own virtual address-space and initialize the SDHC-driver by giving it the base register of the SDHC controller.

After setting up the buffers to read/write to the SD-card and initializing the SDHC-driver, we register the `SDHC-interface` server with the nameserver. We then wait for any incoming RPC-messages to read/write via the default waitset.

9.2.2 Read/Write operations

We wrap the given `sdhc_read_block` and `sdhc_write_block` functions with our own read and write functions. This means we end up with the following API:

- `read_block(index, buf)`, where `index` is the nth block of the SD-card that we want to read and `buf` is the buffer where we read the contents into to return. Before reading the content, we invalidate the virtual address range of our global buffer by calling `cpu_dcache_wbinv_range`, which writes back and the invalidates the address range. We call this

before the read, as optimally we would just invalidate the virtual address range of the buffer, but armv8 does not allow that for EL0. Once the data is read into our global buffer we copy it into `buf` and return.

- `write_block(index, buf)`, where `index` is the nth block of the SD-card we want to write to and `buf` contains the data to be written. We copy the contents of `buf` into our global buffer, then write back and invalidate it to make sure the contents are stored instead of just kept in cache. Then we can write the block to the SD-card and return.

We distinguish incoming read- and write-requests as follows:

- **Read:** The incoming RPC-message has a message-size of `sizeof(int)`, and holds the index of block we want to read. We then send back the read 512 byte buffer as the response message. The client can use the response size to check if the message is invalid - anything besides 512 bytes means the read was not successful.
- **Write:** The incoming RPC-message has a size of `sizeof(int)+512`, and holds the index of the block to be written, followed by the buffer. For convenience we wrap this message in a `struct` for easier passing around. If the write was successful, we send back the character `'d'` with message size 1.

9.3 FAT32 Library Design

9.3.1 Filesystem Type

The filesystem library we implement is able to read SD-cards formatted as FAT32 with a sector size of 512 bytes and 8 sectors per cluster, as mentioned in Section 9.1. We ourselves loosely follow the FAT32 standard, but only support short names. This means that it is possible to name files in a way that is invalid for a proper FAT32 filesystem, or that we do not support character replacement for japanese language (`0xE5`), which would lead to directories not being correctly identified etc. This is mostly due to saving time on not adding additional checks for all the edge cases, or to keep complexity down by avoiding character-swapping or populating multiple FAT-tables, depending on what the BPB-sector entries would dictate.

```
struct fat32_entry {
    char name[12];  // needs to be null-terminated
    bool loaded;  // if its data has been loaded
    bool is_dir;
    bool is_root;
    struct fat32_entry *ddot;  // .. entry
    size_t ref_count;
    uint32_t cluster;
    uint32_t filesize;
    struct fat32_entry *next;  // all entries in the folder
    ↪  except . and ..
    void *filedata;            // filedata if we are a file
};
```

Listing 15: Fat32-entry used to represent the filesystem internally.

Listing 15 shows a filesystem entry node used in our internal representation. `loaded` is used to control if we still need to fetch the content of the entry, e.g. if the directory so far only appears as a child in our internal representation. `is_dir` and `is_root` state if the entry is a (root) directory or a file. The `cluster` gives us the first cluster number holding the contents of the entry, `filesize` tells us how large the `filedata` is if it exists. `*next` stores a pointer to all our children if we are a directory. This ignores the . and .. entries in each folder and starts with the first real child. Each child has `loaded=false`, as we only store basic information like `name` and `cluster`. Lastly, `ddot` holds the .. (parent) directory for each entry, or is `NULL` if the current entry is `root`.

```
struct fat32_handle {
    char *path;
    struct fat32_entry *direntry;
    bool dirty;  // entry has been updated -> writeback needed
    union {
        off_t file_pos;
        struct fat32_entry *dir_pos;
    };
};
```

Listing 16: Fat32-handle used to represent an open handle to a file or directory.

Listing 16 shows a handle to a file/directory entry. It contains the full path to the entry, a pointer to the entry itself, a flag to determine if it is up-to-date with the entry on the SD-card and the current position in the file/directory.

9.3.2 Setup

To initialize the filesystem, we have to call `filesystem_init()`. This does three things:

- `fat32_init()`: Looks up the `sdhc-driver` service on the nameserver and establishes a channel to it. If the service has not been registered yet, it waits a bit and then tries connecting again until it succeeds. This is likely to happen if we initialize our filesystem on `init`, as the initialization of the sdhc-driver is quite slow and thus overlaps with the `filesystem_init()`.
- `fat32_mount_sd()`: This first loads the BPB sector of the FAT32-system on the SD-card and checks if the drive is a valid FAT-drive. It then sets various global variables needed for later computations of reading and writing to the FAT table on the drive, such as the reserved sector count, the FAT table size, the location of the root cluster etc. After knowing the location of the root cluster, we load the directory entry into memory and save it as our root `struct`. When loading a directory into memory, it reads the entry information (name, size, cluster, parent, is-directory.) and creates a child entry into the parent. These children take up little space, as the data they contain is not loaded yet, and they are used as an entrypoint in case they are loaded later on and to traverse the structure.
- `fs_libc_init(state)`: Takes an optional state which we do not use, as we set our root-directory previously already. It sets the function-pointers for the various C filehandling functions. Here we closely follow the approach given by the previous `ramfs`. This means we keep the structure of the functions identical and for every `ramfs` function we create our counterpart that does the same thing with respect to the `FAT32` filesystem.

9.3.3 Opening Files

We give an outline on the process of how a file entry is opened. We first traverse our internal tree-representation of the FAT32 entries to locate the correct entry. If at any point a directory is missing, and it has `loaded=false`, we load the directory from the SD-card, insert it into our internal tree and continue searching for our file in its child entries. Once the entry has been found, we create a handle to it, load its contents, increase the reference count of the file and return. To load the filecontents, we calculate the sector index of the first file-cluster and start loading sectors from that cluster until either the whole cluster or the whole file is loaded. If the whole cluster is loaded, but not the whole file yet, we calculate the sector-index of the FAT table that contains the information about this cluster and get the next cluster

number from that sector. This has some overhead, as the information is only a 32bit value, but we still load and send the whole 512bit buffer via RPC. See Section 9.3.8 for better ways to handle this.

Opening directories works the same way, but instead of loading the filedata, we load and set the children (contents) of the directory.

9.3.4 Creating Entries

Creating a file consists of the following steps:
Checking if it already exists: If it does, we return an error.

Getting the parent directory: This gets a handle to the parent directory, which is the a similar process to how we open a file. We travers our internal tree until we find the parent, load in missing entries and then return the parent handle.

Creating the entry: Once we have the parent, we can create a `fat32_entry` struct that will represent our created file. We set the whole memory area to 0 and set the name (translated to match the FAT32 name format), its parent and if it is a file or directory. We do not assign a cluster, as empty files do not have a cluster assigned. This also means that we do not have to write the file itself to the SD-card yet.

Creating directories is similar as creating files, but as all directories except root have . and .. entries saved, we have to directly allocate a cluster to each new directory and write that back to the SD-card, instead of just updating the parent. This requires us to either re-scan the FAT-table for empty slots each time we need to allocate a cluster, or we keep track of the first free cluster. For this implementation we went with the second approach. This means we can directly assign a cluster to the newly created folder, then update the `first-free-cluster` variable by scanning the FAT-table from that location on for a next free entry. After that we again have to update the parent directory to include the new directory and write that back to disk too.

Using and updating the `next-free-cluster` variable should optimally be an atomic operation, to avoid one cluster being assigned to multiple fat entries at the same time. This could be achieved by putting a lock around it.

9.3.5 Updating Entries

We decided to defer updating a file until the handle to it is being closed. This means once we opened or created a file, we can have multiple writes to it, which only updates the internal representation. We simply modify the in-memory buffers of the filedata and set the `dirty`-flag to indicate that data has changed. Upon closing the handle we check if there have been changes. If there were any, we store the file to disk and also update the parent on

disk, as values such as the filesize, starting cluster etc. are also stored in the parent directory alongside the name of the file.

When storing we have a few cases to watch out for:
Entry shrunk: This may cause less clusters to be used by the entry than before. Thus we check at the end of each cluster if there is still data to be written, and if not we set an EOF value in the FAT-table for that cluster and free any additional cluster belonging to the file by setting the value to 0.

Entry has same amount of clusters: Here we can just write the content of the entry block by block to the SD-card without any changes.

Entry grew: We now have to allocate additional clusters to the file. As mentioned during creating directories, this is done by using the `first-free-cluster` variable and later updating it.

For directories the FAT32 standard allows for two value bits to mark free slots in a directory: 0xE5 and 0x00. 0x00 is used if after that entry all other entries are free slots in the current directory, to avoid having to search through all entries.

We do not use 0x00, but rather use 0xE5. To avoid long fragmentation, every time an entry is deleted, all following entries move up a slot when updating the parent directory. This means we do not have any empty slots between valid entries in a directory and thus do not have to rely and 0x00 to stop early. This also keeps the amount of clusters allocated to a directory minimal as we do not have much free space between valid entries.

To ease writing a directory entry to disk, we have a packed struct with a size of exactly one directory entry (32 bytes), where we copy the values into and then can just directly write back.

9.3.6 Deleting Entries

Removing a file is similar to storing a file that shrunk:
We first get a handle to the file we want to delete and make sure we are the only handle to it. We then remove the file from the parent, update the parent on the SD-card, free the filedata and lastly free any cluster associated with the file itself. We also free the `fat32_entry` structure, as it is no longer referenced anymore.

When freeing the clusters associated to the file, we cannot just set the FAT-table values to 0 and leave the contents of the clusters untouched. This would not matter much for files, but if a directory is assigned to a non-zeroed cluster it may attempt to read entries that not exist due to them not being marked as 0x00/0xE5. Thus we have to set each cluster to 0x00 manually, which is quite slow with the default speed of the block-driver.

We decided that deleting directories should also work on full directories, which means it recursively calls delete on all its entries. This does not follow the default Unix-behaviour of `rmdir`, but was a requirement given, which is why we decided to implement it like this. Otherwise deleting a directory is the same as deleting a file.

9.3.7 Loading ELF Binaries

We modified our `spawn_load_by_cmdline(...)` function to also be able to load ELF-files from our SD-card. We spawn from the SD-card, if the binary name given starts with a '/'. This is then interpreted as the path to the binary. We then read the whole binary into a buffer that can fit the whole file and set the attributes `region_size` and `region_addr` in the spawninfo to the filesize and buffer address. This allows us to then call `spawn_by_spawninfo(...)` as usual.

The main downside to this approach is that due to the rather large size of the binaries (>100kB), it takes a few minutes to load the whole file, which does not really make it a feasible option.

9.3.8 Improvements

Given more time, a first improvement would be proper FAT32 shortname support, handling all invalid characters for the names, utilizing the second FAT-table to make the drive resistant if one of the tables gets invalidated. Additionally additng support for the created-/modified- dates for each entry would be nice. All this would help portability when using files created through barrelfish on other operating systems.

Adding proper thread-safety and atomicity to updates of the `first-free-cluster` variable is a vital improvement that should be quite easy given the provided structure.

It would also make sense to optimize accesses to the FAT-tables: Currently we always load a whole 512 byte block from the FAT-table if we look at one entry, then write back the whole block again. As reading and writing blocks is the main bottleneck for file operations (see Section 13.4), we should cache the entries to the FAT-table, at least if we know that only one domain is working with the SD-card currently.

Another improvement would be that for large files, we defer loading the whole content of the file until we know that we actually need it. The idea being again to minimize unnecessary loads and stores to the SD-card.

Lastly, we think that the given framework for `stat` in `dirent.h` is faulty, as it does not work for directories due to `fopen` only working on files. It then passes a `FILE *` into fstat instead of a directory handle. We tried adding

the necessary steps to lookup the `dirhandle` from the `FILE *` itself, but this would require a rewrite, as we would need to utilize the `fdtab` structures which are located elsewhere and we are not sure if the API can be changed or if it is used somewhere. For implementations relying on the information from fstat, such as getting the filesize or using it in the shell, we use the workaround of reading the whole file to get the size and then rewinding back. This is obviously a suboptimal approach, but it works.

Section 13.4 has some rudimentary performance measurements for the filesystem, but it would be nice to have some more rigorous ones to see any implications between different filesizes, cores and processes more nicely.

Chapter 10

Networking

This chapter covers the development of a simple network stack on our running system. The starting point is the provided ENET driver with the corresponding register regions for the device. The main function of the driver is already polling the receive queue where the processing of the network stack begins. Our network stack receives the unprocessed raw packets from the receiver queue and processes them according to their content. By managing the Ethernet frame itself, we also need to know the MAC address of the destination to which we want to send a packet, which is where the Address Resolution Procotol (ARP) comes into play. In addition, we have built basic support for the Internet Procotol (IP), User Datagram Protocol (UDP) and Internet Control Message Protocol (ICMP) which are covered in this chapter. We first mention here some general information and describe the driver in more detail in the next section. Then we cover the two main parts of our ENET driver, the whole packet processing and the services our ENET driver provides to other user space applications.

Network Byte Ordering We would like to briefly remind you that the received packets are always encoded in the network byte order (big-endian), while our system uses little-endian. The provided helper functions are always used to convert the received values into the host byte order, and when creating the packets to be sent, we convert the used values into the network byte order. So throughout the processing, the values are always converted to host byte order first when they are used or compared. The only exception is the `struct eth_addr`, where we do the conversion from network to host byte order when we need to translate the structure into a `uint64_t`, which is the type of the given MAC address of our board from initialization. The same applies when we need to translate a MAC address of type `uint64_t` into a `struct eth_addr` when building packets.

Host Setup We connect our board directly to the host computer (Arch Linux) via an Ethernet-to-USB adapter. To do this, we need to add an IP address or sub net to the connected device, which already adds a corresponding routing entry for the given sub net. In chapter A we also describe the setup when connecting the board to an Ethernet switch and other useful `netcat` commands for testing.

```
ip addr add 10.0.2.111/24 dev [interface] // setup Host
```

10.1 ENET driver

The supplied driver was almost ready, the only part missing was getting the register regions for the device, which required some changes in the spawning code as we used a separate domain for the driver. The whole process of mapping the device registers has already been explained in Section 8.1.1, but we retype the required region for our driver (`IMX8X_ENET_BASE` and size `IMX8X_ENET_SIZE`) from the `DevFrame` capability to the freshly created `ARGCN` during the spawn code when required. To do this, we added a new function `monitor_spawn_domain` to our monitor, which is basically the same as the function triggered before when spawning a new domain with the corresponding `aos_rpc`, except that in addition `device_base` and `device_size` in the `spawninfo` structure are set to the corresponding device register, where in other spawns these were set to 0, which skips the `ARGCN` setup during spawning. Now, when the ENET driver domain is spawned, the required register region capability is in a known CNode, which is then used to map the frame in the same way as described in Section 8.1.1.

Now the basic driver is ready to go and is already receiving packets via the device queue interface, which has already been wrapped around the sender and receiver queues. But before we go into more detail about how the raw packets are processed, there are a few points worth mentioning about the ENET driver. First we initialize states for the different layers of processing (Ethernet/ARP, ICMP and UDP). We also register services at our nameserver with a corresponding handler function. The provided services are registering for an UDP port, send a UDP packet and the ping command. The provided code in the driver already adds some memory for the receive and sender queue which are mapped into the driver's address space.

10.1.1 TX Queue

When registering the sender and receiver queue, the memory region is registered to the device queue, which in our case consists of 512 entries. For our sender queue, we keep track of an array of 512 entries (`tx_slots` located in `etharp_state`), for which buffers currently belong to the device and which ones belong to our process. In addition we also store the region ID and the

base address from the mapping step. When sending a packet, we get a base address `lvaddr_t` for the next free buffer in the transmitter queue, which is formed from the base address for the sender region with some offset depending on the next free slot. We track `tx_free`, which is the next free slot that we check against the array `tx_slots` if the corresponding buffer is not owned by the device and ready for us. The region ID is then used to create the actual `devq_buf`, which is queued into the sender queue and immediately dequeued afterwards to make the buffer available again. Both operations tries to do it until success or an unhandled error is returned, the handled errors just trigger a retry which could be in the case of enque `DEVQ_ERR_QUEUE_FULL` and for the deque operation `DEVQ_ERR_QUEUE_EMPTY`.

The part we just described includes two places where we use two different mutexes to make it thread-safe. In order to avoid two threads requesting an index for the sender queue and writing afterwards to the same place in the buffer we used a mutex to make the call `get_free_slot` thread safe. The second mutex is placed around the `enqueue` operation on our sender queue which caused some trouble during the stress test and performance measurement.

After registering the sender and receiver queue, we enter the state where the driver polls the receive queue and processes each received packet using the valid base address and length (`valid_length`). To get the base address of the new packet, we use the base address of the receiver queue from the mapping and add the buffer offset (offset relative to the start of the region) and the `valid_data` (offset into the buffer where valid data is present). Now we have the address to the outermost header of our received packet and start to deal with the actual data of the received packets, which is described in the next section which is futher split up depending on the actual layer.

10.2 Packet Processing

If you pass the packet to another layer, you can simply add the header length of the current layer to the address and pass the new address to the next layer pointing to the correct location in the buffer for the next layer. An important note at this point is that the processing of packages should never block and should be quietly fast so that we can process the next package. The only critical section is when we actually send out a packet in response to a received packet, but this should never block unless our sender queue is full or due to a missing ARP cache entry. The second case should not occur in our friendly environment, but in the case of spoofing the source IP address, this would cause an ARP broadcast that is unlikely to receive a response and cause a timeout. This would prevent us from processing any more packages in the meantime. Another critical point here is the UDP data forwarding

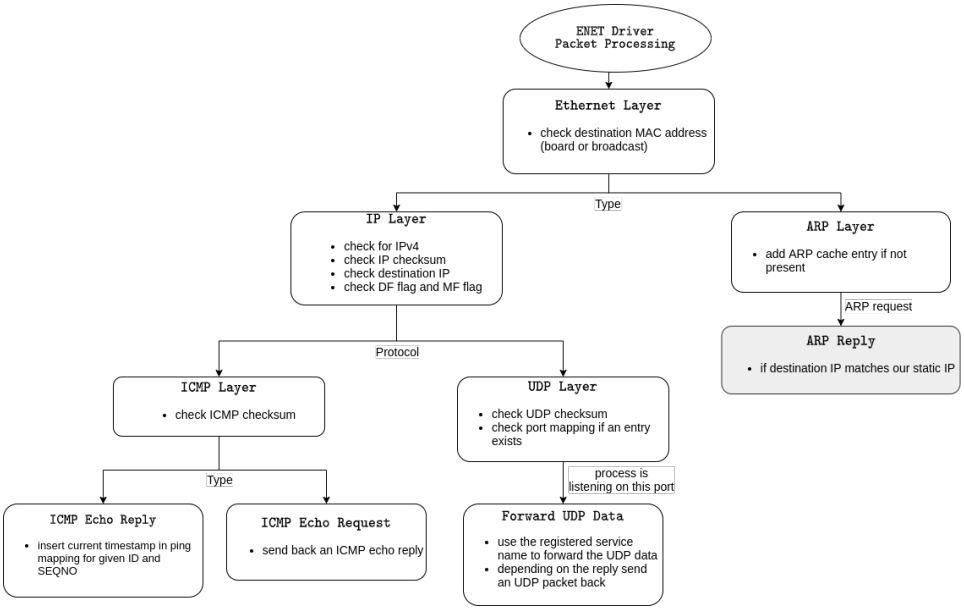


Figure 10.1: Flow diagram of the actions taken by the ENET driver thread polling the receiver queue if a packet is successfully dequeued.

to user space applications discussed in Section 10.4. Figure 10.1 provides a big overview about the packet processing where each step is descriped in the next few secions.

10.2.1 Ethernet Layer

The first header we encounter when processing a packet is the Ethernet header. We check whether the destination MAC is the correct one from the board or all 1s, which corresponds to an ARP broadcast. If neither case applies, we simply return and ignore the packet. Depending on the Ethernet header type we either face an underlying IP packet or ARP packet which is processed next by increasing the address by the length of the Ethernet header.

For sending a packet we have built helper functions for each layer taking the base address as input and a `genoffset_t` which is increased by the size of data added to the packet and used later as `valid_length` in the `devq_buf`. Each helper function requires some additional input depending on the layer, so for the Ethernet packet we need to specify the type and the destination IP address, the second being used to look up the destination MAC address in the ARP cache, which is then used for the Ethernet header. Specific to the Ethernet packet is the additional length for the frame check sequence (CRC) at the end of the Ethernet frame after the actual payload.

10.2.2 ARP Layer

When we receive an Ethernet packet of type ARP (`0x0806`), we add the corresponding source IP to source MAC address mapping to our ARP cache, regardless of whether it is an ARP request or response (`opcode` in ARP header). For replies, there is nothing else to do, while for requests, we send out an ARP reply telling the sender our link layer address (MAC address) if the destination IP matches the static IP of our board (`10.0.2.1`). Every time we send out a packet, we mentioned just before that we look up the destination MAC address in the ARP cache, in the given scenario here we have just added the corresponding entry to the cache, but if we send out a packet to a destination IP address for which we do not have a corresponding ARP cache entry, the lookup will trigger an ARP broadcast and constantly check the ARP cache for two seconds to see if the desired entry has been added in the meantime. If not, we return an error saying that the ARP request failed and we could not resolve the specified destination IP. The noteworthy thing about the ARP broadcast is the Ethernet header destination MAC address, which is set to all ones, the target hardware address (THA) field of the ARP header, which is set to all zeros and the destination IP is set to the one you want to query. It is also worth mentioning here that during the process of receiving and processing a packet, we will never encounter the case where we actually send out a packet for which the corresponding ARP cache entry is not yet available, so the timeout scenario during the ARP broadcast will never slow down our process of receiving and processing the packets.

ARP cache The ARP cache described earlier is just an instance of a hash table from the provided collection libraries with `128` buckets that store the translation from IP address to MAC address, where the IP address is the key and the retrieved element (MAC address) is of type `struct eth_addr`.

10.2.3 IP Layer

Back in the Ethernet layer, we could also receive a packet of type IP (`0x0800`) where the Internet protocol processing starts. The IP layer is the first one where have a checksum field in the header, the provided implementation of the checksum algorithm computes the 1's complement sum over the same fields including the checksum field and then returns the bit wise complement which is `0` if the checksum is correct. The same helper function can be used to compute the checksum when sending out an IP packet except that we set the checksum field to `0` before computing it. In the next step we check if the destination IP address matches our static IP, for all checks we immediately return if the check fails and ignore the received packet.

Sending IP Packets The IP packet contains not only the IP header but also IP data, which is, for example, an ICMP header and the corresponding ICMP data. When sending an IP packet, we need to specify the actual length of the IP data payload to form the IP header. The checksum is only an integrity check for the IP header and not for the actual IP data.

IP Offset While we should simply detect and discard incoming fragmented packets, the way we handle this is a bit more elaborate for what we want to achieve. When sending a UDP packet from the host to our board, the DF flag is set if the actual data is small enough not to exceed the MTU. Otherwise, the actual data is split and IP fragmentation is used to deliver the packet through the link layer. We assume that we do not need to support long messages where IP fragmentation is used, yet we wanted to support the utility to ping the host from the board. The problem with this is that the echo (ping) responses sent from the `linux` host do not set the DF flag. Therefore, we have implemented the following IP fragmentation rule:

- When DF flag is set, process the packet.
- Now we probably deal with a fragmented IP packet, we only process the packet if the more fragments flag (MF) is not set. So we are able to process the ICMP Echo replies for pinging sent by the host.

Using this method influences if we forward fragmented UDP packets to the UDP layer which is discussed later in Section 10.4. Now we continue the processing depending on the protocol type of the IP header where we currently support ICMP and UDP.

10.2.4 ICMP Layer

This is the first layer where not only the current header is checked, but also the ICMP data payload is processed. The ICMP header also contains a checksum, which is checked in the same way as the IP checksum. When forming an ICMP packet, the process for calculating the checksum is the same, but it is worth noting that for `IPv4` this is calculated from the ICMP message starting with the type field of the ICMP header (without the IP header), whereas for `IPv6` it is preceded by a pseudo header. But don't get too excited, unfortunately we don't support `IPv6` at the moment.

The rest of the procedure depends on the ICMP header type, we only support echo request and echo replies. The echo request is quite simple, we just send back an ICMP packet with the same ICMP data and the destination IP address set to the source address of the received packet. The echo response corresponds to an executed ping command in our shell and is explained in more detail in Section 10.3.1.

10.2.5 UDP Layer

The UDP layer has already been mentioned briefly and is now explained in more detail. UDP processing is very similar to ICMP processing, whereby the IP packet contains IP data that is now interpreted as UDP header with the corresponding UDP data. The UDP checksum is optional for `IPv4`, i.e. if the checksum field of the UDP header is 0 (unused), processing continues, otherwise the checksum is checked. To check it, the UDP checksum needs a pseudo IP header (length 12) in front of the UDP header, which mainly contains the IP address for source and destination. When sending UDP packets, the checksum is currently not calculated.

UDP Ports When we process UDP packets, what happens depends on the destination port. Suppose our system has just started and we are already receiving UDP packets. At this point, the processing ends and we discard the received UDP packet. The reason for this is that we are actually only forwarding packets to an application that has previously registered on a UDP port. We again use a hash table collection to store the translation from UDP port to a service name registered for the given port. Based on this service name, we forward the UDP data through our name server infrastructure, which requires that the application registering for a UDP port specify a valid service name. Registering for a UDP port is a service our ENET driver provides to user space applications and is described in more detail in Section 10.3.

Depending on the response message of the previously invoked service, we either send back the same UDP data, send back custom data supplied in the response message, or do not respond at all. This covers the last point of our packet processing journey, where we return to the point polling the receiver queue to process the next packet.

10.3 Provided Services

This section describes the services that our ENET driver provides for user-space applications.

10.3.1 PING command

While we do not support terminating shell commands, we have implemented a ping command that sends out a fixed number of ICMP echo requests about 1 second after the last response has been successfully received. The ICMP data contains only a timestamp, which is obtained by the provided timer functions in `aos/systime.h`. Each ping command corresponds to an id and each request has a sequence number (seqno). After sending, we constantly

check a collection `ping_mapping` that maps a key (combination of id and seqno) to a timestamp if the given id and seqno combination is present. If the entry is present, we have received the corresponding response from the host. This is also all we do when we receive an echo reply message, we simply add the id/seqno combination to the `ping_mapping`, where the added timestamp is the time we received (processed) the packet. If we do not receive a response for one second, we assume that the corresponding device is not able to respond to the ICMP packet and print "Request Timed Out". This case is different compared to an unreachable network, which would already be detected when our ARP broadcast times out. The described scenario would lead to the message "ping: connect: Network is unreachable".

Performance In our first implementation, we added the timestamp contained in the ICMP packet to the collection when receiving an echo reply and estimated the actual RTT when the corresponding collection entry was present. This approach resulted in an estimated RTT of about `120 ms` for a ping echo request/response where the ARP entry is already present. Since the collection operations should not be included in the RTT estimate, we changed the added timestamp when processing an echo reply to be the receive time, which is then used for the RTT estimate. The resulting RTT is now about `57 ms` when pinging the directly connected host and seems to be a more reasonable calculation method. This small change reduced the average RTT, but we are still much slower than the average RTT of `0.6 ms` when pinging the directly connected router from the host machine. Although when we ping the board from the host computer we get an average RTT of about `0.28 ms`, it seems that processing the received packets and sending out the echo response works very fast, while sending ping requests and receiving the echo response takes much longer. The issue here was the running thread for the ping command checking if the collection entry is present in a busy loop. So we yield the thread after sending out the echo request to allow the polling driver to process a response immediately in the perfect scenario. This results in a satisfactory RTT of about `0.28 ms` for a ping request and the reception of the corresponding response. The ping command also measures the time used for the entire command, which includes the second we wait between subsequent requests. Nevertheless, the timing between several ping executions is quite stable, except for the first execution, which takes about `160 ms` longer, due to the fact that our system starts up with an empty ARP cache. So before we send the first ping request, we check whether the corresponding ARP entry is available or whether we need to send an ARP broadcast. This is done to avoid adding the time required for the ARP broadcast and the reception of the ARP response to the first RTT estimate.

10.3.2 Register to listen on a UDP port

When an application wants to receive UDP packets destined for a specific port, it requires registering a service at the name server which is able to handle `struct udp_msg` messages. These messages contain the actual UDP data and length, the source IP address and port, and the destination port. The reason for all this information is that the same structure is used by the provided ENET driver service to send out a UDP packet from user space applications described in the next subsection. The actual operation when registering for a UDP port is quite simple, it just adds the translation between UDP port to the corresponding service name to the `port_mapping` collection. Some important notes about the services registered to a UDP port are discussed in the next section where we describe our UDP Echo server in more detail.

10.3.3 Send UDP data

The last service is quite straight forward, it just sends out a UDP packet to the provided destination IP address/port pair. A big problem here is that the user space applications can only send out one UDP packet at a time because we have a thread processing the requested services which is blocking until the previous handler returned. The performance implication of this is shown in Chapter 13.

10.4 UDP Echo Server

An example for the previously described port registering is our `udpServer` acting as a UDP echo server outputting the UDP data and trigger the driver to send the same UDP data back by setting the correct flags in the service response. When starting such a process, you can pass an argument for the port number on which it should listen. It then registers a service at the name server, which is later used by our ENET domain to forward the UDP packet. Note that we have a process that polls the receiver queue. So when we forward a UDP packet through our name server infrastructure, we cannot process any more packets until the invoked service has returned. So if you are designing an application where you want to receive UDP data, it is important to design the callback function for the registered service so that it returns without much overhead. This is of course not perfect, creating a detached thread to forward the UDP packets in the ENET domain would be a more robust solution, but we wanted to avoid creating threads during packet processing and creating a thread for each each UDP packet we want to forward is also not a solution. When redesigning this part we would add a queue where we add packet we want to forward to user space and one thread checking this queue if there is anything to forward. By using a hash table for port-to-service translation, it is currently not possible for multiple processes

to register on the same port, which is roughly the equivalent of a UDP socket.

IP fragmentation Our UDP echo server is now running and working, but unfortunately when sending messages that exceed the MTU, IP fragmentation is used on the host side that sends the UDP data. On the receiver side we ignore the packets where the more fragments flag (MF) is set as already mentioned. The last packet which includes also the UDP header could be forwarded in theory, but the UDP checksum check obviously fails, which would require reassembling the different IP fragments. Also we would just be able to forward the UDP data contained in the last packet while dropping the previous received fragmented packets. The way we handle IP fragmentation allows us to accept the ICMP echo responses without the DF flag set to support pinging the host, while large (>MTU) UDP packets are not supported.

10.5 ENET Driver Performance

For performance measurements we designed different scenarios where we measured quantities depending on the goal of the simulation. The detailed discussion about the results is located in Chapter 13 where we just provide a short overview here:

- Measuring the Upstream Bandwidth achieved by sending packets directly from our ENET domain compared to the achieved Upstream Bandwidth when multiple user space application use the registered service in our ENET domain to send out a UDP packet.
- Latency comparison between the estimated RTT time of ping messages handled directly in our ENET domain compared to the required time difference in Wireshark when sending out UDP packets from a `python` application on our Host until Wireshark observed the corresponding UDP Echo reply triggered by `udpServer` acting as a UDP Echo server.

Chapter 11

Nameserver

A nameserver allows services to register themselves under a known name, so that clients can look up services on the nameserver and then contact the individual services directly.

Many nameserver operations are also available as shell commands (see Section 8.2.4).

11.1 Nameserver Process

The nameserver is a special process that runs on core 0 and stores mappings between service names and a unique service identifier. It can be contacted through RPC, to lookup, enumerate, register, or deregister services or validate service names.

The registered services are stored in both a fixed size array, indexed with the identifier as well as as hashtable. The hashtable allows for quick lookups of services by name while the array makes it easier look up a service by id and find a free id to give to a new service.

A service is also uniquely identified by a name, but since we don't want to send around the service name for all requests, each service gets assigned a unique 16-bit unsigned integer (`srv_id_t`) by the nameserver with id 0 being reserved to detect non-initialized service ids. This leaves room for 65535 services that can run concurrently on the system. In practice the service is identified by both the PID it runs on and its service ID because the service ID does not have any routing information attached to it. A unique ID separate from the PID is needed because we want to support running multiple servers in the same domain. We chose to make the service IDs unique across the whole system instead of just per process because it makes managing them a lot less complex.

A service name must be between one and 127 characters (inclusive) and may only contain the following characters:

- Alphanumeric characters (`[a-zA-Z0-9]`)
- Dash (`-`)
- Underscore (`_`)
- Slash (`/`)
- Period (`.`)

In addition, service names are case *insensitive* and will be lowercased in nameserver.

11.2 Bootstrap

No process has a direct connection to the nameserver, except for `init` on the core that launched it (core 0). To contact the nameserver, a process first has to contact `init`. For this, we implemented a specialized nameservice RPC message that can be sent to the init process on the same core which then forwards it to the nameserver or to init on the other core depending on which init was contacted.

This forwarding works in the same way as forwarding things like spawn requests between cores: Init receives the request, parses it and passes the message buffer to `aos_rpc_send_recv_data` to send it to its destination (either init on the other core or the nameserver itself). It then returns the reply it received as if it generated the reply itself.

11.3 Nameserver API

Processes that want to communicate with the nameserver should use the API defined in `nameserver.h` and do not have to manually craft nameservice RPC messages.

11.3.1 Server API

For servers that want to provide a service can simply call `nameservice_register` with a service name, a callback function and additional data to be included in the callback.

The nameservice library registers a callback on the RPC listener channel for every new process and when receiving service messages, it forwards it to the correct registered service.

```
struct nameservice_chan {
    const char *name;
    srv_t srv;
};
typedef struct {
    aos_pid_t pid;
    srv_id_t id;
} srv_t;
```

Listing 17: Structure of service handler.

A service can deregister itself by calling `nameservice_deregister`. If a process exits, `init` also deregisters all services on the process.

11.3.2 Client API

Clients that want to send data to a service, can get a handle to the service (`nameservice_chan_t`) by calling `nameservice_lookup` or `nameservice_list` (`nameservice_list` should be preferred to `nameservice_enumerate` because it directly returns a service handle). The structure of the handle is shown in Listing 17.

To send requests to a server, the client can then use `nameservice_rpc` with the obtained channel handle. Note that capability transfer is not supported and both `capref` arguments must be `NULL_CAP`.

The format of the message and response is completely defined by the service and does not follow a fixed format.

11.4 Service Routing

While our message passing stack does allow for P2P channels between arbitrary processes (even across cores), we do not use this and route all requests through the `init` process on the same core.

All service messages sent through `nameservice_rpc` and sent back from the server receive handler are encapsulated into `AOS_RPC_SERVICE` and `AOS_RPC_SERVICE_REPLY` messages. The `AOS_RPC_SERVICE` message contains both the PID and the service ID in its header and allows the init process to route the request to the correct service on its own core or to init on another core if the service is on a different core. The forwarding functions in the exact same way as routing nameserver messages.

```
nameservice_chan_t chan;
errval_t err;
do {
    err = nameservice_lookup(NAME, &chan);
    barrelfish_usleep(10000);
} while (err == LIB_ERR_NS_NOT_FOUND);
```

Listing 18: Waiting for a service to become available.

11.5 Using the Nameserver in our System

Both the filesystem as well as the network stack register themselves with the nameserver. The filesystem uses it to read and write blocks on the SD card and the networking process registers a server for sending pings, one for sending out UDP packets and one to register a listener on a port.

The processes listening on a UDP port themselves also register a service and send its name to the networking process so that it knows where to forward incoming UDP packets.

We do not use the nameservice for requesting ram caps because it already runs on `init` and does thus not need to register itself anywhere since all processes already know its location and because we already need a way to allocate ram caps before actually starting processes. If one really wanted, one could solve this by setting aside memory for bootstrapping the memory server, the nameserver and all the ram caps they request once started up and once all of them are running, switching over to using the memory server. However, this does massively increase the complexity of system startup and won't bring many benefits since all ram requests will be routed through `init` anyway and the routing overhead will be comparable, if not worse, than managing the memory in init. For the same reason we also did not extract the serial driver and LED driver into separate server processes.

11.6 Waiting Until Service is Available

If both server and client for the same service are started close to each other, it becomes possible that the client will try to look up the server before the server had a chance to register it. This will lead to a `LIB_ERR_NS_NOT_FOUND` error.

In that situation, the only solution is to continously look up the server until it appears. Listing 18 shows a simple way to do this without spamming the channel.

Ideally there would be a blocking lookup request where the nameservice only

replies once the service is registered. Unfortunately, due to the nature of message passing system, this is almost impossible. If the nameserver only replies once the server is registered, the forwarding logic in `init` would also block on the receive call. In our system, sends and receives together must be atomic, so while `init` is waiting, the entire channel is blocked and once the server tries to register, we get a deadlock because the forwarding logic blocks the channel waiting for the server to register itself and the server waits on the channel to be unblocked.

There is a way around this: The client creates its own server that the nameservice could contact once a service becomes available and the client would just have to loop locally until it receives a message for that server. This adds a big overhead on service lookup and so we decided to just go with the loop and sleeping.

Chapter 12

Capabilities revisited

12.1 Sharing memory

During the project we opted to have the `init` processes act as the memory servers for the cores they are running on. When the new `init` process for core 1 is created a URPC message is sent to it, which contains a bootinfo struct that has a single empty memory region in the beginning, followed by mmstrings and the module capabilities (needed for spawning processes). When `init` on core 1 starts running, it can read bootinfo to forge capabilities.

By having this divided memory management between cores, we were able to minimize the number of capabilities that needed to be sent (and forged) between cores. Consequently the first goal of this milestone was to enhance how this singular capability exchange worked. The ram capability we send is set to be owned, by core 1 and all the other capabilities are just sent as foreign ones.

To facilitate this the first message core 1 receives from core 0 was extended. During the construction of this message the `struct capability` of all entries are gathered into an array and the owner of each, set according to the above described ownership model. Finally this array is attached to the end of the message and on the receiving side core 1 can create capabilities from these (and also set the owners according to the region type of the entry).

Later the memory server part of `init` on core 1 was changed, so that for requests that exceed the size of 2 * PTABLE_ENTRIES * BASE_PAGE_SIZE bytes it would request the ram cap from core 0. This was mainly to show how our system could handle a single memory server and how we could share capabilities. In this model core 0 would give out ram capabilities, which would then be owned by the destination core.

12.2 Capability operations

As above mentioned, now cores would own their ram capabilities, and those capabilities would have a remote copy on core 0. This means that if any operations (delete, revoke, retype) were performed on them, there is now a need to synchronize across cores. To facilitate these operations one aos_rpc call was added for all processes:

```
/**
 * \brief Request init to delete a remote capability
 * \arg croot the root cnode of this domain
 * \arg addr the capaddr of the cap
 * \arg level the level of the cap
 * \arg op the operation to be performed (e.g. DELETE)
 */
errval_t aos_rpc_monitor_cap_op(
    struct aos_rpc *rpc,
    struct capref croot,
    capaddr_t addr,
    uint8_t level,
    enum cap_op op
);
```

However, the two `init` processes now need a way to communicate with each other and these communications should only happen between init processes. Therefore in a file only accessible by `init` another two calls are added:

```
/**
 * \brief Request RAM from init on core 0
 * \arg size the size of the requested RAM cap
 * \arg ret the received capability
 */
errval_t cap_init_ram_transfer(
    struct aos_rpc *rpc,
    size_t size,
    struct capability *ret
);
/**
 * \brief Transfer cap back to init.0
 * \arg cap the capability to delete/transfer back to init 0
 */
errval_t cap_remote_del(
    struct aos_rpc *rpc,
    struct capability *cap
);
```

12.2.1 Delete

When an `init` process receives a request for a delete operation that means that the regular delete operation returned a `SYS_ERR_RETRY_THROUGH_MONITOR` error code. This could be either because the capability is owned by this core, but has a remote copy and therefore needs synchronization or the deletion of the capability cannot be done in a bounded number of steps.

If the capability can be moved we simply transfer the ownership of it to the other core (core 0) and delete our local copy. Since in our system, every capability that leaves core 0 is a ram capability core 0 simply needs to retrieve the capref and give it back to the memory manager. Here a system that keeps track of capabilities is needed or one could create a copy with `monitor_copy_if_exists()` (a copy is guaranteed, since that's how the other core got it). This copy can then be used to give back the memory to `mm`.

If it cannot be moved then the last owned copy needs to be deleted. For this we utilise the delete stepping framework provided. All other cleanup tasks (e.g. delete the resulting capability) are performed in the callback function that gets called after the deletion has run to completion.

If there were any remote capabilities then for every non-movable type the other core(s) would need to be instructed to delete their copies before the last copy can be deleted. This situation does not present itself in our implementation and is therefore omitted.

12.2.2 Revoke

For every revoke call a `SYS_ERR_RETRY_THROUGH_MONITOR` error is almost guaranteed to be thrown (unless some other error occurs) and so the involvement of the monitor is necessary. This is due to the fact that while revoking, new copies/descendants could be created (and it could have in theory and infinite amount of steps).

Locally revoking capabilities is done by marking all the nodes that need to be deleted and then using the delete stepping framework to simply just delete all copies.

Remotely revoking capabilities would be very similar except with the other core performing the revoke on their CSpace as well.

12.2.3 Retype

Unfortunately, due to many reasons this operation of this project remained unfinished. In theory it would be simple since the data of the retype would need to be simply sent over to core 0, which would just return a yes or no answer and then the retype could just happen.

There were some messages added to facilitate this between `init` processes:

```
/**
 * \brief Check if capability can be retyped
 * \param cap the capability to be retyped
 * \param out the result signalling if it can be retyped or not
 */
errval_t cap_can_retype(
    struct aos_rpc *rpc,
    struct capability *cap,
    bool *out,
    gensize_t offset,
    gensize_t objsize,
    size_t count
);
```

But these never ended being used/called.

Part IV

Our OS

Chapter 13

Performance evaluation

This chapter presents some experimental measurements taken from our system and also the interaction of our network stack with a given host machine. Each section describes a specific scenario and what quantities we measure and how we measure them. The interpretation of performance measurements of a self-built system is quite difficult without suitable reference values. Nevertheless, we tried to reflect on the results, finding some quite interesting ones. However, it must be said that we do not think that our system is particularly strong in terms of performance. But hardly anyone achieves the desired goal when they do it for the first time, and in this case, it is sometimes even unclear what the goal is.

13.1 Networking

13.1.1 Upstream Bandwidth

The first scenario is about the ENET driver sending potential in terms of achieved upstream bandwidth. The approach is quite simple: we start four threads and each one is sending out `100k` UDP packets. The packets consists of 446 bytes UDP data resulting in a total frame size of 500 bytes. Each send involves claiming a free `tx slot`, filling out the corresponding header fields (Ethernet, IP, and UDP), and the corresponding `memcpy` of the provided data string into the actual buffer. Moreover, it includes a collection lookup for the MAC address used by the Ethernet layer and the checksum computation for the IP header. The prepared packet is then ready to be pushed onto the sender queue which also includes waiting until we dequeued it successfully to make the buffer available again. Figure 13.1 shows the achieved upstream bandwidth using Wireshark on the host machine. We achieve just under 80 Mbit/s, while the Toradex board has a 100 Mbit Ethernet port.

Saturating the network connection capacity on 100% seems to be quite

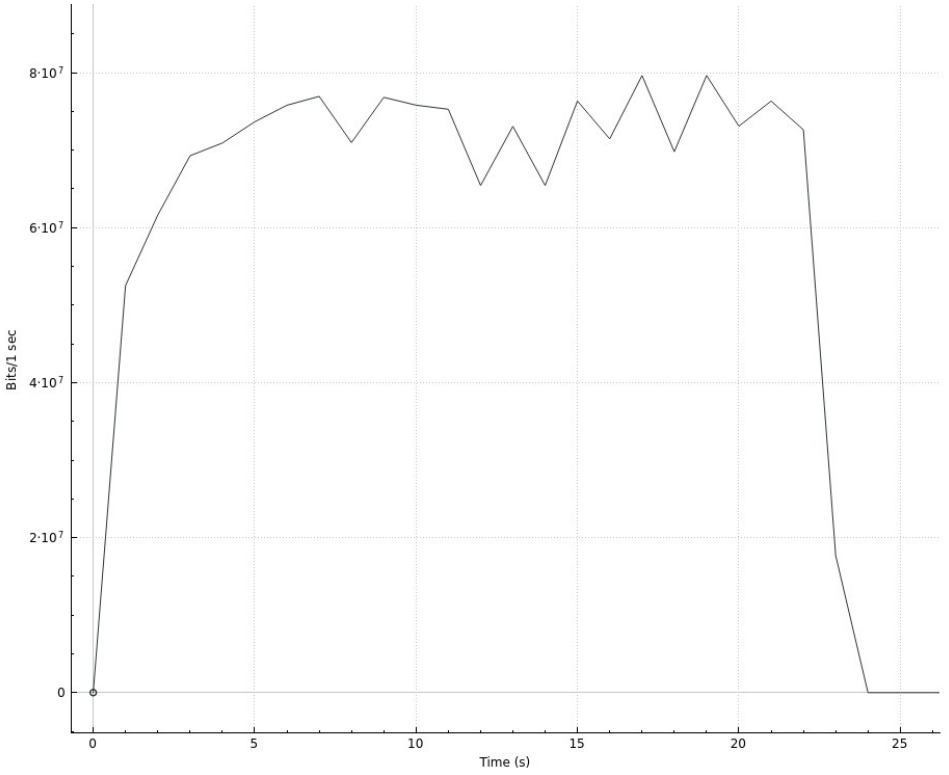


Figure 13.1: 400k UDP packets with a total frame size of 500 bytes sent by four threads in our ENET domain. The data was collected and processed in Wireshark.

challenging. A big problem is managing the sender queue. For preparing the buffer, we only need a lock for receiving the free slot index. This seems unavoidable unless we allocate some fixed slots to each thread. However, we still have a mutex when we actually add the buffer to the queue. Removing the mutex leads to strange behaviour in the `enet_devq` backend, which is quite hard to debug when it only occurs under heavy load. If we were to change our current sender queue management to not wait until we have successfully dequeued the previously requested buffer, we could probably fill up the sender queue and get closer to the actual hardware-limited bandwidth.

While adding more threads which send out UDP packets, we discovered an issue where threads exited returning the error `LIB_ERR_ARP_REQ_FAILED`. This was caused by the initial timeout of one second. Some threads checked the collection for the appropriate MAC address, which is not initially present, and were therefore descheduled. By the time they were rescheduled, the timeout period had already expired, resulting in the previously mentioned error. The error still occurs when adding about 8 threads sending UDP traffic

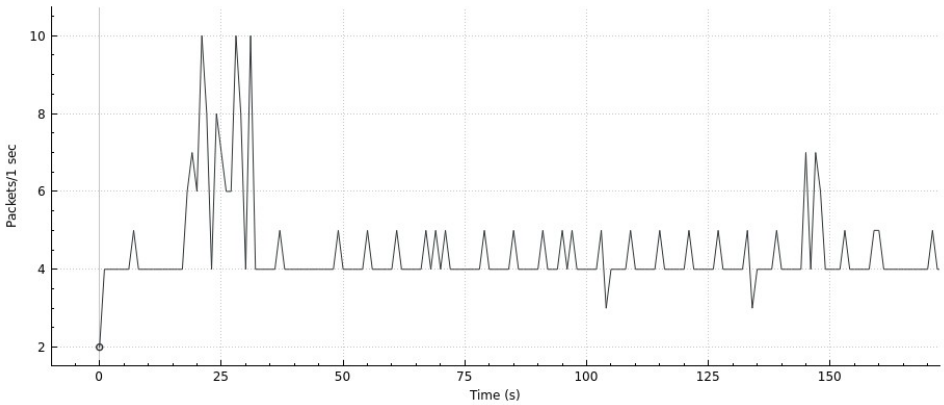


Figure 13.2: Multiple domains sending out UDP packets of a total frame size of 500 bytes. The received packets per second were measured with Wireshark.

using a timeout of two seconds, but we want to avoid setting this timeout any higher. The reason for this is that we want to avoid threads checking the collection over and over again, where this case should only be triggered if we actually do not receive an ARP response.

We also added a process which spawns ten `enetSender` processes which use the provided `enetSendUdp` service registered by the ENET domain to send out UDP packets to our host. Each `enetSender` triggers one send after the other, where each packet has the same total frame size of 500 bytes as in the previous experiment. During this setup each send requires a `memcpy` to copy the UDP data (446 bytes) into the message used for `nameservice_rpc` and a second one in our ENET domain to copy the data in the actual buffer. Figure 13.2 shows that we achieve a peak performance of 10 packets per second, which corresponds to using 40KBit/s bandwidth. This is much less compared to the scenario where we actually send UDP packets directly from our ENET domain without using our built infrastructure for message passing between domains. So to improve the usable bandwidth for user-space applications, we would need to speed up our message passing and also the way we handle services in our ENET domain. Currently, the ENET domain has a thread that handles the services and does not process the next service request until the previous handler has returned. This blocking leads user space applications only being able to sequentially place one packet after the other in the sender queue, which slows things drastically down.

13.2 User Space Latency

In Section 10.3.1 we have already discussed the `ping` results that attempt to estimate the RTT. Since our `ping` command triggers a registered service

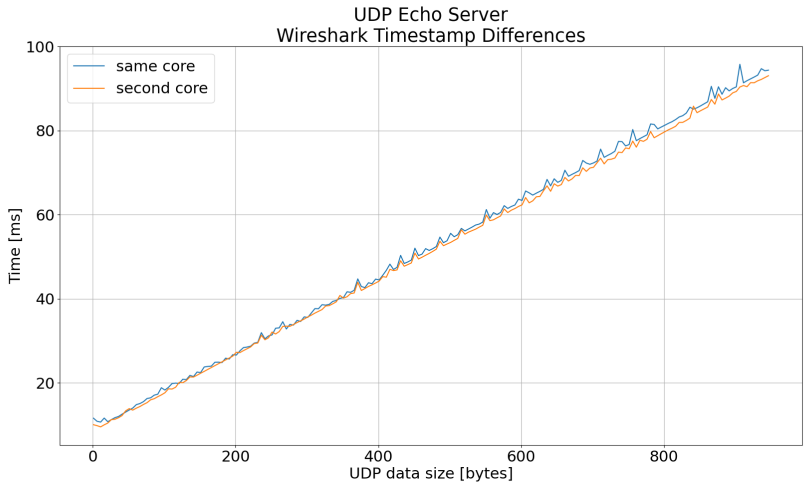


Figure 13.3: Time difference between the timestamp of Wireshark when sending the UDP packet to the board and receiving the UDP Echo server reply for various sizes. The blue line corresponds to the measurements for the UDP Echo server running on the same core as the ENET driver while the other line corresponds to the UDP Echo server running on the other core.

in the ENET domain that actually sends the ICMP echo requests and waits for the response, it does not forward any data to the user while measuring the RTT. In order to get a better picture of our network stack's latency, we take a look at the time required to send a UDP packet from a `python` application on our host until we receive the Echo reply sent from the running `udpServer` on our board. The required time is calculated using the Wireshark timestamps of the observed outgoing and incoming packets. Although our system is in an almost idle state during these measurements, we still added a one second delay in our `python` script before we send out the next packet to avoid interference. We tested this for a UDP echo server running on the same core as the ENET driver as well as a UDP server running on the second core. After each packet we increase the UDP data size by 5 bytes.

The measured times heavily depends on the UDP data size as shown in Figure 13.3: packets carrying only a few bytes of data arrive back at the host after about `12 ms`, while echoing larger packets takes linearly more time. The only difference depending on the data size is that occurs during processing packets and echoing them is the size for the `memcpy`, and the amount of data which is printed by our `udpServer`. The linearly increasing time for larger packets seems therefore reasonable to us.

Recall the RTT estimate of the ping command is about `0.28 ms`, which is about `0.17 ms` when calculating the timestamp difference observed by

Wireshark. When we compare those low RTT estimation to the required time where we actually process the data in user space on our board, the latency is much higher. We have already discussed the reason for this at the beginning of this section: in both cases – answering ICMP echo requests or sending our own – the entire process takes place in the ENET domain and we are not required to send messages between different domains, or at least not while we are actually measuring the time to estimate the RTT. This is no longer the case in this scenario, where the ENET driver actually sends messages with LMP or even UMP when the `udpServer` is running on the other core. Therefore, the significant slowdown seems to originate in the message passing. Section 13.3 further investigates this.

13.3 Message Passing

Measuring the user space latency for the network stack made us curious about the performance of the message passing library. Therefore, we created two small processes, one acting as a server and the other as a client. The client uses a service registered by the server, which involves sending a string that the server returns. We incrementally increased the actual string size sent by our client by 500 bytes per step, starting with 0 and going up to 8 Kbytes. For each size, we sent the same message 20 times to see any variation in the estimated round-trip time measured on the client. We repeated the described procedure for each of the four combinations of server/client spawn location (i.e., server on core 0, client on core 0; server on core 1, client on core 0, etc.).

Figures 13.4 and 13.5 display the average for all 20 repetitions for each size. In addition, the bars for each dot show the minimum and maximum estimated round-trip time. Even without showing the distribution, we observe that the average is usually close to the observed minimum. This is because the high latency measurements were usually only single outliers. We think those outliers are probably caused by a slot allocator that is being filled or a pagefault. The fact that this behaviour occurs at regular intervals supports this hypothesis.

Figure 13.4 shows that we achieve an average latency below `5 ms` for every measured byte size when both server and client are running on core 1. In comparison, we observe a slightly higher latency when both processes are running on core 0. Still, most of the averages remain below `10 ms`. We explain this with the fact that our second core is – core 0 – mostly idle. Moreover, we observe that there are higher outliers on core 0. We expect this to be due to the mentioned problem that `init` on core 0 runs more tasks, which can cause delays when they perform resource-intensive tasks.

Next, we compare Figure 13.4 to running server and client run on different cores (shown in Figure 13.5). When the server runs on core 1 and the client

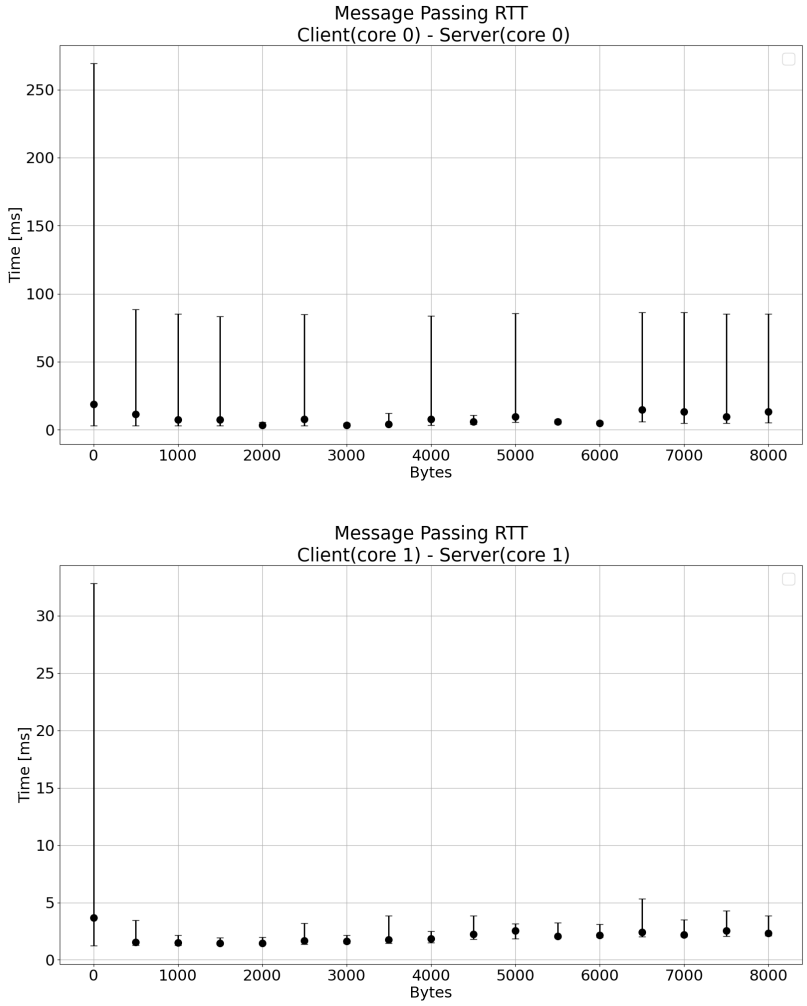


Figure 13.4: Estimated Client-Server round-trip times for Client/Server located on the same core.

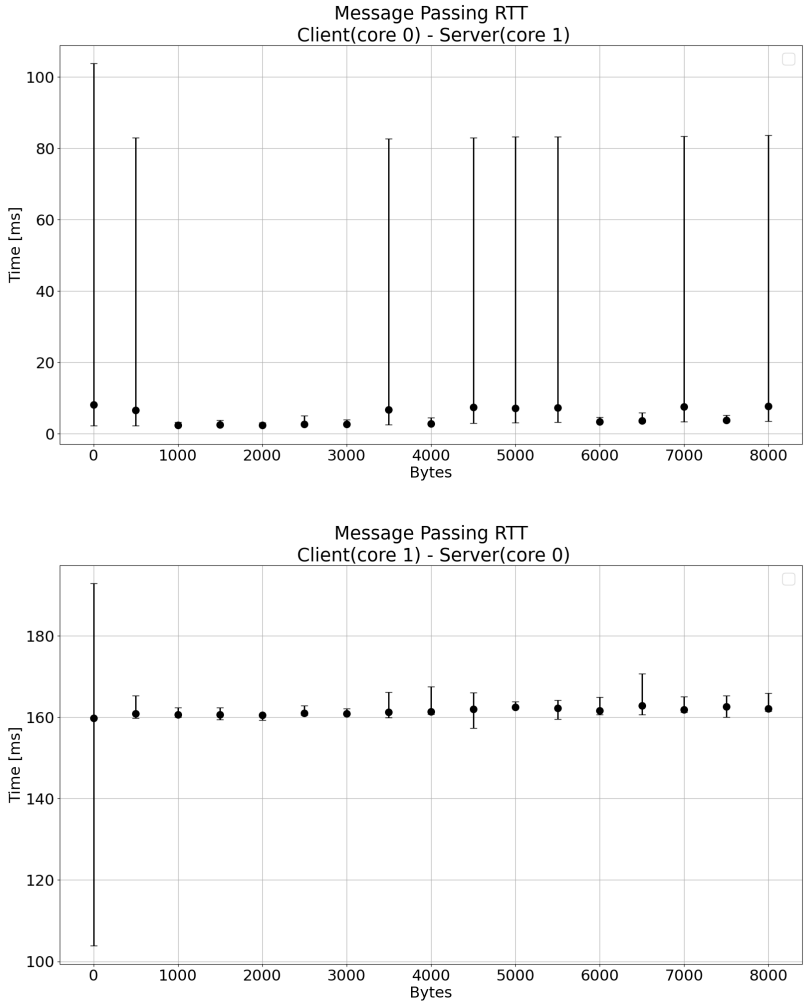


Figure 13.5: Estimated Client-Server round-trip times for Client/Server located on different cores.

on core 0, we achieve about the same average as when both run on core 0. However, it gets interesting when the server is running on core 0 and the client on core 1: suddenly, we measure a more or less constant RTT around `160 ms`. This is much higher than any other average we have observed so far. Since the processes run on different cores, the transfer also involves UMP transfers.

Before investigation the cause of this behaviour, let us first recall our inter-core communication. Although our UMP transfers have a high throughput – as we take advantage of the cache coherency protocol – they still have a high latency. This is because we use the polled channels in the waitset implementation and this polling runs only when the process is (de)scheduled. This long reaction time only applies to the listener (i.e., the server in our case), because when waiting for a response over UMP, we constantly poll the shared buffer.

One thing that could explain the drastic difference between the the client running on core 1 and the server on core 0 and the other way around is that the waitset channels are polled less frequently on core 0. This is because there are more dispatchers scheduled on core 0. This potentially causes high latency when the server runs on core 0 and is contacted by a client on core 1 over UMP, because the server only receives a message when the waitset is polled.

The same thing does not happen when both run on core 0, because then only LMP channels are involved. Those channels are not polled. Instead, the kernel directly wakes up threads when messages arrive.

13.4 Filesystem

13.4.1 SD Block Driver

As our current implementation of the `read-` and `write`-functions utilize a lot of buffer copying - each time we read and write we copy the whole 512 byte buffer - we were curious how much slows down the overall performance. Additionally we also invalidate at least 512 bytes of cache before each read or write, which could also have negative impact on the performance.
Read performance: Reading a block of 512 bytes takes roughly half a second (461ms). From that, copying the buffer and invalidating+writing back the cache each take only 0.00026% of the time (1us).
Write performance: Writing a block of 512 bytes also takes roughly half a second (470ms). From that, again copying the buffer and writing back the cache only take 1us each.

This clearly shows that the majority of the time is spent in the actual read-/write-call of the given SDHC-driver, and that the overhead of buffer-copying

is insignificant. The slow reading-performance has implications on loading binaries from disk: A simple sample program with the size of 500kB would need to do 1000 individual reads, which would take almost 8 minutes, which is quite a bad performance.

13.4.2 File Operations

Here we give a short overview of some of the execution times that can be expected for creating a new file, deleting a small file and deleting a large file.

Creating file Creating an empty file in the root directory takes 5.3 seconds. Here we can nicely see the amount of time it takes to update the directory the file is contained in, as creating an empty file does not allocate any cluster for the file itself. The only thing we update on the SD-card is the root directory itself. Creating an empty file in a subdirectory of root takes 6.2 seconds. This takes slightly longer as creating an empty file in the root directory, which is as expected. This is likely caused because we first have to fetch the contents of the subdirectory, whereas root is already loaded. In both cases we then update the current directory we are in.

Deleting a file Deleting a small file with contents "Hello world!" in the root directory takes 11.5 seconds, deleting a large file (10kB) in the root directory takes 26.8 seconds. We can see the large overhead of updating the root directory on the SD-card and zeroing out the whole cluster allocated with the small file, as the large file is 10x larger, but takes less than 3 times as long to execute.

Chapter 14

Testing

Testing an OS is hard. It will never have the same conveniences as modern, high level languages, built on top of existing OSs and VMs, can provide, but we still sought a good testing framework. We wrote test processes that demonstrate functionality, which relies on our IPC implementation. We also tried to write our own unit test framework and use that where IPC wasn't involved. This framework proved to be more useful in the beginning, but was also a great guard against regression.

14.1 The unit testing framework

To write something, where we can just declare test cases that run at least semi-automatically we used macros. A lot of macros. The whole framework is just a library of macros.

In our design we ended up having:

- Test groups: These need to be run manually. It is a macro that translates to a function, which then can be called anywhere the build allows it to be. In theory test groups would also be a unit of testing having the same buildup and teardown calls, but we never ended up implementing this and opted to use helper macros, since they could do more anyway.
- Test cases: Test cases go inside test groups, and translate to blocks of code. These run automatically with all other tests in the same group. They can either pass or fail in which case they also print the failing assertion and the location of the assertion.
- Assertions: Assertions go inside test cases and confirm truth values. They are the fundamental building blocks. We implemented a range of

assertions starting with assert equal, larger through to assert errval or assert page fault.

A simple test case in our system would look something like this:

```
TEST_GROUP(paging, {
    TEST_CASE(paging_unmap, {
        int *addr;
        TEST_PAGING(BASE_PAGE_SIZE, addr);
        paging_unmap(get_current_paging_state(), addr);

        // We expect a page fault here
        ASSERT_PAGE_FAULT(true, addr[1] = 666);
    });
})

void run_test_suite(void) {
    RUN_TEST_GROUP(paging);
}
```

14.2 Testing modules on top of the core

While the unit testing framework described in the previous part was used to ensure that the basic operations in our system continued to work while we build things on top of the core, it was difficult to integrate the later parts into the same testing framework. Therefore, we added some applications inspired by the supplied `spawnTester` and `memeater` that use the added functionality to make sure they work as intended. For some parts it didn't make sense to have an isolated process just for testing, like the shell. But other parts have some kind of test process like `serverTest` for the name server or `enetSender` for the ENET driver. It is also worth noting that some applications we have added, for example the `udpServer` acting as an UDP echo server, also help us to test the system continually using the network stack and also the name server infrastructure. Testing the shell was implicitly always included after the integration by us currently using it.

References

[1] Timothy Roscoe and the ETH Barrelfish team. *Advanced Operating SystemsSpring Semester 2021*. Apr. 2021.

Appendices

Appendix A

User Guide

In order to run our system the setup steps are exactly the same as described in the AOS book. After successfully setting up and booting the corresponding image on the Toradex board, our system is ready for use. The start-up process takes some time while initializing the file system for our shell. We briefly mention the behaviour in case of missing hardware in our system and the required setup to have a working network stack.

Missing SD card or Ethernet Cable If no SD card is inserted an error (`SDHC_ERR_CMD_TIMEOUT`) is returned when starting the `sdhc_interface` process indicating the missing SD card. The shell still continues to work but without support for the files system commands. For the Ethernet cable, the ENET driver will not complete its initialisation and therefore will not register the services offered. The ENET driver does not crash and when you connect the cable after booting, auto-negotiation is completed and the driver works as intended. While the system continues to work, if you unplug the cable again, we obviously don't receive a packet, but the driver continues to queue packets in the send queue when e.g. `sendUdp` is used, just without the packet actually being sent.

SD-card Setup The filesystem assumes that the SD-card is formatted as a FAT32 filesystem directly, with the first block of the card containing data relevant to the filesystem. This means we cannot have the filesystem on a separate partition on the SD-card. Optimally the card is formatted as given in the book, e.g., with `mkfs.vfat -I -F 32 -S 512 -s 8 /dev/xx` for linux machines. The filesystem is mounted on `/sdcard`. As only the FAT32 filesystem is running, this means that any paths that do not start with `/sdcard` are invalid.

Networking Setup For the network stack to work properly it depends on how you wish to connect the board. If we plug the Ethernet cable directly to the host it requires us to add an IP address or a sub net to the corresponding interface with respect to static IP we assign to our board (`STATIC_IP` in `etharp.h`) as illustrated in Listing 19 line 1. This also adds a route for the given sub net which is required to get things working. If we connect the board to a switch it requires no more setup as long as the `STATIC_IP` set in `etharp.h` is free and located in the local sub net. The latter setup adds the ability to also `ping` the router.

```
1 ip addr add 10.0.2.111/24 dev [interface]
2 nc -l -u [port]
3 nc -u [ip] [port]
```

Listing 19: Useful commands to setup host and using `netcat` on the host to receive UDP data.

When we reach this point, we are in the state where our system is running and everything we have added to our system should work. The command line interface (shell) is now the way to interact with our system, the `help` command is a good starting point showing all available commands with a short description. The main functionality is provided by the file system commands, the `ping` command and the ability to start other processes with the `oncore` command. Each command also includes a help page by typing `cmd -h`, but for further user guidance about the shell we refer to the section 8.2.

A.1 Network Stack Interaction

In order to use our UDP echo server (`udpServer` process) we start the `netcat` command on line 3 in Listing 19 on our host for the `STATIC_IP` we use and a port number, the default port used when starting a `udpServer` without a specified port is 7. If we now enter a string the UDP packet gets sent to the specified IP addres/port destination. Before we start our UDP echo server the packet gets dropped during process, bus as soon as we start the `udpServer` for the corresponding port the string gets printed to the shell and we receive the echo reply on the host machine. The `ping` command does not need any explanation, the best thing is just to try it out, though it will only work for machines on the local network. You can also send UDP data from the shell using the command `sendUdp` but make sure you start a listener on the host beforehand using the command on line 2 in Listing 19 with the port you want to listen on.

A.2 Shell

This section contains a general user guide for our shell. Section A.2.1 describes a few quirks of our shell. Section A.2.2 contains the help pages for all available commands.

First, some basic information on the shell. To get a shell, spawn the module `spawnShell` with a call to `monitor_spawn_module` (this is done in our default configuration).

A.2.1 Pitfalls

In this section, we list some common pitfalls and quirks of our shell:

- **Limited relative paths:** we only support a very limited version of relative paths. The single and double dots for the current and parent directory are *not* supported. However, the current working directory is prepended to paths in case they do not start with a leading slash.
- **Attached/detached spawning:** by default, `oncore` spawns processes detached from the shell. This means, they will compete with the shell for the serial console. Due to line multiplexing, it is possible that some output of a command does not appear when you already entered characters of the next command. They will appear after you pressed enter.
- **`ls` for files:** the command `ls` is only supported for directories, and *not* for files (it would be easy to add, we just did not find the time to make it general).
- **Commandline arguments for `oncore`:** To pass additional command-line arguments to processes spawned with `oncore`, you have to wrap the name of the executable and the command-line arguments in quotes. For example for spawning program `foo` with arguments `bar` and `baz`, one would run `oncore "foo bar baz"`.
- **File operations are slow:** Due to an unoptimized block driver most file operations are quite slow. For example reading a 500KB file (e.g. when launching an ELF executable from the filesystem) takes between 5 and 10 minutes.

A.2.2 Command Help Pages

The following list links to listings with the help pages for all existing commands:

- **cat:** Listing 20

- **cd:** Listing 21
- **demo:** Listing 22
- **echo:** Listing 23
- **exit:** Listing 24
- **help:** Listing 25
- **info:** Listing 26
- **led:** Listing 27
- **ls:** Listing 28
- **mkdir:** Listing 29
- **nslist:** Listing 30
- **nslookup:** Listing 31
- **nsvalidate:** Listing 32
- **oncore:** Listing 33
- **ping:** Listing 34
- **ps:** Listing 35
- **pwd:** Listing 36
- **rm:** Listing 37
- **rmdir:** Listing 38
- **run:** Listing 39
- **send:** Listing 40
- **sendUdp:** Listing 41
- **write:** Listing 42

```
Help for command 'cat':
cat [options] path [path, ...]

Usage:
Output content of file on the given path.

Positional arguments:
Path(s) to files(s) for which cat displays their content

Optional arguments:
-h, --help Prints this help page; no arguments
```

Listing 20: Help page of `cat`

```
Help for command 'cd':
cd [options] path

Usage:
Change current working directory to the given path.

Positional arguments:
New path for working directory

Optional arguments:
-h, --help Prints this help page; no arguments
```

Listing 21: Help page of `cd`

```
Help for command 'demo':
demo [options]

Usage:
Demonstrate working user-level threads and serial channel
↪  multiplexing.

Optional arguments:
-h, --help   Prints this help page; no arguments
-i, --input  Demonstrate input (getchar) multiplexing; no
↪  arguments
-o, --output Demonstrate output (putchar) multiplexing; no
↪  arguments
```

Listing 22: Help page of `demo`

```
Help for command 'echo':
echo [options] text [text...]

Usage:
Prints its arguments to the shell.

Positional arguments:
Text to print to the shell. Takes 0 or more arguments

Optional arguments:
-h, --help   Prints this help page; no arguments
-e, --escape Interpret backslash escaped sequences; no
↪  arguments
```

Listing 23: Help page of `echo`

```
Help for command 'exit':
exit [options]

Usage:
Exit this shell.

Optional arguments:
-h, --help Prints this help page; no arguments
```

Listing 24: Help page of `exit`

```
Help for command 'help':
help [options] command

Usage:
Prints available commands and their help pages.

Positional arguments:
Displays the command's help page

Optional arguments:
-h, --help Prints this help page; no arguments
```

Listing 25: Help page of `help`

```
Help for command 'info':
info [options]

Usage:
Prints information on the system and the current runtime
↪ environment.

Optional arguments:
-h, --help     Prints this help page; no arguments
-s, --shell    Keyboard shortcuts supported on this shell; no
↪ arguments
-c, --coreid   Current core's id; no arguments
-t, --threadid Current thread's id; no arguments
-p, --pid      Get PID of this process; no arguments
```

Listing 26: Help page of `info`

```
Help for command 'led':
led [options] on/off

Usage:
Turn LED4 on the Aster Carrier Board on/off.

Positional arguments:
Turn LED4 on/off

Optional arguments:
-h, --help Prints this help page; no arguments
--on       Turn LED4 on; no arguments
--off      Turn LED4 off; no arguments
```

Listing 27: Help page of `led`

```
Help for command 'ls':
ls [options] path [path, ...]

Usage:
Lists the content of directories.

Positional arguments:
Path(s) to folder(s) for which ls lists the content

Optional arguments:
-h, --help  Prints this help page; no arguments
-l, --list  List elements vertically; no arguments
-s, --stats Output name, type, and size in a table; no
↪  arguments
```

Listing 28: Help page of `ls`

```
Help for command 'mkdir':
mkdir [options] path

Usage:
Create a directory at the given path.

Positional arguments:
Path to new directory

Optional arguments:
-h, --help Prints this help page; no arguments
```

Listing 29: Help page of `mkdir`

```
Help for command 'nslist':
nslist [options] query

Usage:
List all active services.

Positional arguments:
Optional regex search string

Optional arguments:
-h, --help Prints this help page; no arguments
```

Listing 30: Help page of `nslist`

```
Help for command 'nslookup':
nslookup [options] name

Usage:
Lookup service through nameserver.

Positional arguments:
Name of the service

Optional arguments:
-h, --help Prints this help page; no arguments
```

Listing 31: Help page of `nslookup`

```
Help for command 'nsvalidate':
nsvalidate [options] name

Usage:
Validate a nameservice name.

Positional arguments:
Name to validate

Optional arguments:
-h, --help Prints this help page; no arguments
```

Listing 32: Help page of `nsvalidate`

```
Help for command 'oncore':
oncore [options]

Usage:
Run a module on the specified core (default: current core).

Optional arguments:
-h, --help     Prints this help page; no arguments
-c, --core     Specify core id on which the application should
↪  be run
-a, --attached Spawn application attached to the current shell;
↪  no arguments
```

Listing 33: Help page of `oncore`

```
Help for command 'ping':
ping [options] destination

Usage:
Sends ICMP echo requests to the given IP and print replies.

Positional arguments:
destination IP address

Optional arguments:
-h, --help Prints this help page; no arguments
```

Listing 34: Help page of `ping`

```
Help for command 'ps':
ps [options]

Usage:
Displays currently running processes.

Optional arguments:
-h, --help Prints this help page; no arguments
-c, --core Specify core id for which the processes are
↪  displayed. Use 'all' or '-1' for all cores
```

Listing 35: Help page of `ps`

```
Help for command 'pwd':
pwd [options]

Usage:
Print current working directory.

Optional arguments:
-h, --help Prints this help page; no arguments
```

Listing 36: Help page of `pwd`

```
Help for command 'rm':
rm [options] path

Usage:
Remove the file at the given path.

Positional arguments:
Path to the file to remove

Optional arguments:
-h, --help Prints this help page; no arguments
```

Listing 37: Help page of `rm`

```
Help for command 'rmdir':
rmdir [options] path

Usage:
Remove the directory at the given path.

Positional arguments:
Path to directory to be removed

Optional arguments:
-h, --help Prints this help page; no arguments
```

Listing 38: Help page of `rmdir`

```
Help for command 'run':
run [options] path

Usage:
Execute shell commands stored in the given file.

Positional arguments:
Path to file to execute

Optional arguments:
-h, --help Prints this help page; no arguments
```

Listing 39: Help page of `run`

```
Help for command 'send':
send [options] name

Usage:
Send a message to the named server.

Positional arguments:
Name of the server to which a message should be sent

Optional arguments:
-h, --help   Prints this help page; no arguments
-s, --string String to send
```

Listing 40: Help page of `send`

```
Help for command 'sendUdp':
sendUdp [options] <data>

Usage:
Send UDP packet incl. the given data to the specified IP and
↪  port

Positional arguments:
data to send

Optional arguments:
-h, --help    Prints this help page; no arguments
-a, --address destination IP address; required
-p, --port    destination UDP port; required
```

Listing 41: Help page of `sendUdp`

```
Help for command 'write':
write [options] path

Usage:
Write or append string to file.

Positional arguments:
Path to file that should be written.

Optional arguments:
-h, --help   Prints this help page; no arguments
-e, --escape Interpret backslash escaped sequences; no
↪  arguments
-s, --string String to write to file
-f, --force  Overwrite file if it already exists; no arguments
-a, --append Append to an existing file; no arguments
```

Listing 42: Help page of `write`