Advanced Operating Systems

Hacking the Barrelfish OS

Andrin Bertschi
bandrin@ethz.ch

Raphael Eikenberg
reikenberg@ethz.ch

Christian Leopoldseder
cleopolds@ethz.ch

Loris Reiff
lreiff@ethz.ch

4th June 2020

```
Barrelfish CPU driver starting on ARMv8 (BSP)
kernel 0: ARMv8-A: 4 cores in system

.............................
....._....___..____..___....
...../ \  / _ \/ ___|| |.| |...
...../ _ \| |.| \___ \| |.| |...
..../ ___ \ |.| |...) |  _  |...
.../_/...\_\___/|____/|_|.|_|...
.............................
Welcome to AOSH..............
AOSH Operating System Shell.....
.............................

aosh >>> nslist server
There are 5 services matching query 'server':
serverinit
serverblockdriver
serverfilesystem
servermonitor0
servermonitor1
serverprocess
serverserial

aosh >>> cat myfile2.txt
File size is 70
I love deadlines. I like the whooshing sound they make as they fly by.

aosh >>> oncore -f arp
Querying ARP cache...
00:14:2d:64:13:a4 - 10.0.0.2
00:25:96:12:34:56 - 10.0.0.1
1C:96:AE:84:4A:E9 - 10.0.0.3

aosh >>> ip
10.0.0.2

aosh >>> pwd
/sdcard/team/

aosh >>> ls members
.
..
bean
chris
eikendev
liblor
```

Contents

1. Memory Management

> *Manually managing blocks of memory in C is like juggling bars of soap in a prison shower: It's all fun and games until you forget about one of them.*
>
> — anonymous

Since milestone 1 needed to be done individually, we chose one of our implementations of the memory manager. As a group we improved a few details about it. This chapter explains how that implementation works.

1.1. Internal Memory Representation

The `init` dispatcher uses the `mm` library (libmm) to manage its physical memory. Initially, the `init` dispatcher receives a list of free blocks in memory from the CPU driver in the form of capabilities, which are added to the Memory Manager. We call these initial blocks of manageable memory *major memory blocks*. The basis of the internal data structure of the memory manager is a doubly linked list where each node represents a block of memory. Each node stores a node type, a base address, a size, and the RAM capability of its underlying major memory block. For now, there are two types of nodes: *free* and *allocated*. Initially, when the major memory blocks are added to the Memory Manager, one node per block will be created and they are inserted in the order of their base addresses. In general, the list of nodes will always be kept in order of the base addresses of the memory regions the nodes represent and nodes never represent overlapping memory regions. To ensure that, the Memory Manager also checks for overlaps when major memory blocks are added. Furthermore, a node must always have a non-zero size.

1.2. Memory Allocation

When the new memory is allocated using `mm_alloc()` or `mm_alloc_aligned()`, the Memory Manager is supposed to provide an unused RAM capability of sufficient size and possibly a given alignment.
As a first step, the Memory Manager will select a free node that is large enough to accommodate the requested size of memory, also considering a possible increase of size to conform to the specified alignment. If the Memory Manager cannot find such a node marked as free it means it ran out of memory.
The selected node might be very large, which means if it were to be returned directly a lot of space would be wasted. This is why as a second step, it will split the selected node up into at most three new nodes:

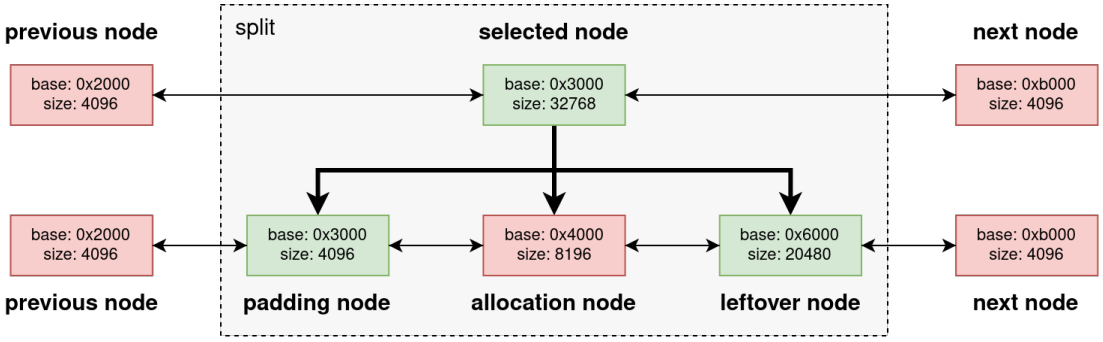

Figure 1.1.: Example of a node being split into a padding node, an allocation node, and a leftover node. The allocation node represents the memory region that will be returned to the caller as the requested memory.

1. a *padding* node,
2. the *allocation* node,
3. and a *leftover* node.

The allocation node will have a base address conforming to the specified alignment and at least the requested size. The padding node may be necessary to ensure the alignment, and the leftover node will represent the excess memory of the selected node. Note that the padding node and the leftover node will only be created if they are of a non-zero size. Added together, those nodes represent a memory region of exactly the size of original selected node. They will be inserted into the doubly linked list in place of the selected node and they will inherit the capability of the major memory block stored in the selected node. An example of such a split is visualized in figure 1.1.
The implementation actually reuses the selected node as the allocation node and just changes its base address and size, so only the padding and leftover nodes would be truly newly created.
The capability of the selected node will be retyped to the region represented by the allocation node. The allocation node will subsequently be of type allocated, while the padding and leftover nodes will remain as type free.

1.2.1. Allocation Strategy

The Memory Manager has to pick a free block that has enough capacity for the requested size. There are different allocation strategies available to select such a free block, that each result in a different fragmentation pattern. We decided to use the *worst-fit* allocation strategy, which means we select the largest block that is still free. We chose it because it is very simple, runs in linear time, and its resulting fragmentation is not too bad. We also considered the *best-fit* allocation strategy, but—arguably counter-intuitively—this strategy leads to worse fragmentation patterns than worst-fit.[1] Lastly, we considered

[1]Source: grading session for milestone 1

implementing the *buddy* memory allocation strategy since it has even better performance[2], but the implementation was postponed because the worst-fit strategy works fine in our system.

1.3. Memory Deallocation

When a block of memory is no longer needed, it can be returned to libmm, which allows it to be reused for other needs. To find the exact node that has to be freed in its internal list, libmm iterates all entries, until a match is found. After the block was found, it is marked as free.
Further, libmm has to ensure that contiguous free blocks are merged into a single free block when possible. If we never merged free blocks, it would become a lot harder to find a free block for allocation.
Hence, when a block is freed, libmm checks if it can be merged with its neighbor blocks. Of course, neighboring blocks with different major memory blocks will never be merged.

1.4. Slab threshold

For maintaining the internal list in libmm, a slab allocator is used. Specifically, the slab allocator provides libmm with memory in case new list nodes are inserted.
Since these list nodes are essential for certain functions to properly work, we need to ensure that the slab allocator always has free nodes available. For this, we introduced a function that first queries the number of free slabs in a slab allocator, and conditionally refills it with new memory. This function must then be called wherever we need to ensure the margin.
Further note that the margin needs to be higher than the amount of slabs used in a single allocation. If the allocation consumes all slabs available in a slab allocator, there are no more slabs left for refilling it.

1.5. Optimizations and Improvements

Fragmentation To decrease fragmentation we could compare different allocation strategies in our application setting and their impact on fragmentation. However, this makes more sense at a later stage of the project as it is difficult to predict the allocation pattern. Similarly, one could read up on research about allocation and fragmentation.

Performance The linked-list could be replaced by a asymptotically more efficient data-structure, like for instance a red-black or AVL tree. As for the different allocation schemes we could then perform measurements and compare it to the literature.

[2]Source: “Operating Systems - Internals and Design Principles” by William Stallings

2. Processes, Threads and Dispatch

> *Realize you won't master data structures until you are working on a real-world problem and discover that a hash is the solution to your performance woes.*
>
> — Robert Love

2.1. Paging

In milestone 1, each team member wrote a simple paging implementation, which only supported a small subset of virtual addresses. Thus, we had to reimplement the paging code in this milestone in order to spawn new dispatchers.

2.1.1. Shadow Page Table

To keep track of the current page table state one has to maintain a shadow page table which keeps a consistent state with the actual page table. This is necessary as we can not read the page tables from the MMU.
There are some degrees of freedom when it comes to the data structure used to maintain such a shadow page table. Given that the mapping of memory should be fast, we need a data structure with fast lookup, but also with fast insertion time. It is obvious that data structures such as linked lists and trees are not well suited for this task. An array or hash table is a way better choice in terms of performance, as both provide constant time lookup and insertion.

Array As mentioned arrays are an obvious choice for a shadow page table due to constant time lookup and insertion. However, there are also other factors we wanted to consider, namely, memory footprint. Given that ARMv8 has a 4-level page table and generally only a subset of addresses will be mapped, some tables will end up with only a few entries. Thus if we allocate a 4KiB array for each table we will end up with a huge memory overhead.

Hash Table Hash tables come with the benefit of constant time lookup and insertion on average and use less space. In our opinion this makes them well suited for a shadow page table. However, if a page table has many entries the likelihood of collisions increases. Collisions decrease the speed of hash tables significantly.

Hash Table + Array Since we only have bad performance with almost full hash tables, it makes sense to use hash tables for use-cases with few elements. When we expect many entries we might as well use an array, since the only drawback of the array in this setting was its memory footprint. We reasoned that page table L0-L2 won't

have as many entries as L3, where we expect that often all 512 entries will be used. Therefore, we used hash tables for the page tables L0 to L2 and an array for the L3 page table. Furthermore, we only allocate memory for page tables on demand.

2.1.2. Tracking Free Address Regions

With regards to milestone 4 and some code that uses `paging_regions` we opted to additionally keep track of the virtual address regions that are used in a separate data structure. While this can be incorporated into the previous data structures, we chose not to do so for several reasons. For instance, hash tables are not well suited to find a free region of the appropriate size. This action is important when a user wants to map a frame without naming a specific virtual address. Another benefit of choosing a dedicated data structure for this task is that we can easily distinguish between keeping track of free blocks in memory and the decision when to create a new page table, which will be especially beneficial in milestone 4. Lastly, the decision is also partly motivated by the fact that the `paging_regions` structs were used in existing code and we didn't want to break existing code. As we will learn later, this code is not and will not be relevant for us. An alternative rather "hacky" approach we quickly rejected is to reserve a subset of virtual addresses which could be used for mappings where the virtual address is not specified. This should be possible due to the large address space of ARMv8. That way we wouldn't really need another data structure, as keeping a pointer to the free region is enough. However, this implies that another address range is not explicitly usable by dispatchers. Moreover, the alignment has to be the same for all addresses as fragmentation will occur otherwise. Furthermore, with regards to milestone 4 we will need some sort of tracking anyway. Also, this approach is not really compatible with an `unmap` function.
With the aforementioned goals, we opted for a linked list approach. We keep track of the address range with a dedicated module, which is basically a doubly linked list similar to the approach in libmm. This decision is motivated by the fact that the implementation is straightforward and we are less likely to introduce unnecessary bugs, which would make us miss the deadline. We also considered generalize the code of the memory manager, which rejected at this point due to time constraints. The linked list adds some cost to the paging code, as we now have a linear in the number of regions lookup in the paging function. There is certainly room for improvement, when it comes to runtime efficiency. As one can see in section 4.3.1, this module is subject to rewrites.

2.1.3. Putting it all Together

At this point our paging system works as follows. A user can either map a frame via one of the following functions.

```
errval_t paging_map_fixed_attr(struct paging_state *st, lvaddr_t vaddr,
                               struct capref frame, size_t bytes, int flags);

errval_t paging_map_frame_attr(struct paging_state *st, void **buf,
```

```
                    size_t bytes, struct capref frame,
                    int flags, void *arg1, void *arg2);
```

In the first case, the caller has to specify the wished virtual address itself. In `paging_map_frame_attr` a free virtual address is chosen automatically. Internally, it will use `paging_map_fixed_attr` afterwards. If a dispatcher calls `paging_map_frame_attr` with the arguments to map a frame of some size[1] (e.g. `16384` bytes), the linked list that keeps track of the free virtual addresses (section 2.1.2) is walked through. The address of the first free address space large enough is then passed to `paging_map_fixed_attr`, with the other arguments. Here the virtual address linked list is walked through again to see whether the passed `vaddr` is indeed free. This is necessary as `paging_map_fixed_attr` is, as previously mentioned, also exposed to the user. If the address region is unused, it is marked as used. At this point the shadow page table walk happens. Naturally, the mappings can spawn over multiple page tables. This is solved by looping until the whole size is mapped. If the mapping spawns over multiple page tables, the tables are rewalked again completely (see figure 2.1). This decision is motivated by the impression that this results in cleaner code.

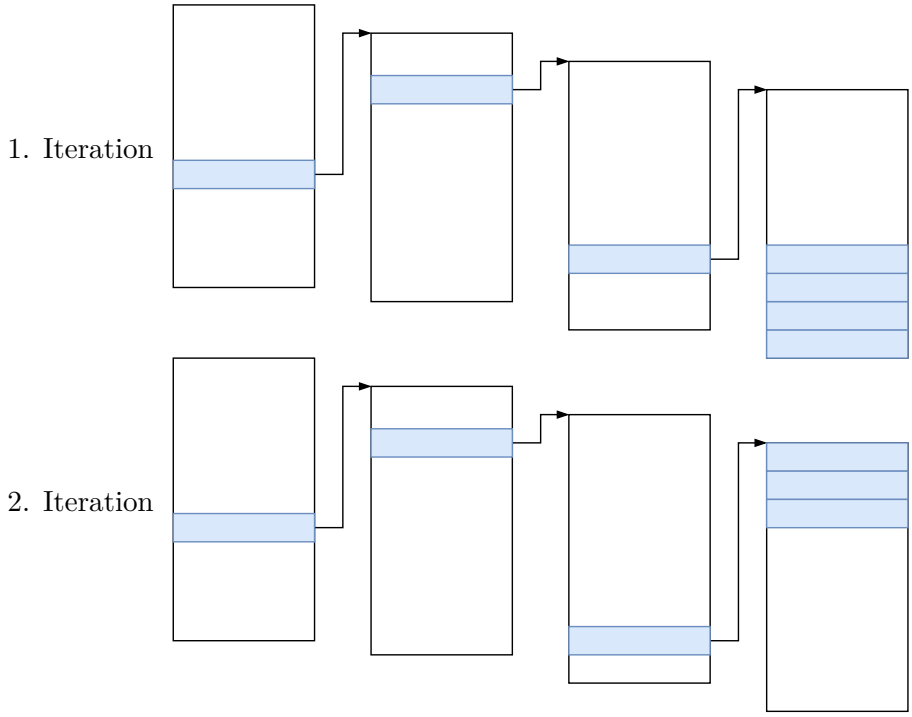

Figure 2.1.: Shadow page table walk for address regions spawning over multiple tables

As explained in section 2.1.1 page tables level 0 to 2 are implemented as a hash table. Page tables in level 3 are a block of memory, i.e. an array. The tables are created along the way if they don't exist and the corresponding mapping capabilities are created and

[1]Note, the size is always round up to a multiple of the `BASE_PAGE_SIZE` to avoid fragmentation.

eventually the mapping to the user passed frame is instantiated. From here on, the user can access the frame over the virtual address according to the flags passed.

2.1.4. Technical Challenges

Paging code is generally expected to just work as it is such a central part of a modern operating system. The abstraction of a paging system is rather clear. The devil lies in the implementation details. During development we introduced a number of bugs, which were not always trivial to find. The origin of the bugs often did not show right away, as the symptoms appeared in other components due to the very nature of the memory abstraction. Therefore, it is of uttermost importance to test the code in various ways. Needless to say that writing extensive tests is a challenge in itself.

2.2. Spawing A Dispatcher

Until this point, we were only running a single dispatcher. To become more flexible, and to address a major feature that operating systems usually provide, it is now time to implement a way of spawning new dispatcher.
In this section, we refer to the parent dispatcher, i.e., the dispatcher that wants to spawn another dispatcher, as the *parent*. Similarly, we use the term *child* to refer to the to-be-spawned dispatcher.

2.2.1. Borrowing Memory

A new concept for us was how Barrelfish thinks of dispatchers in regards to memory management. However, understanding this fundamental is an essential ingredient to start a new dispatcher.
In general, a dispatcher "receives" some memory when it is setup. This is necessary, otherwise there would be no memory to load the code from, nor to maintain a stack. During its lifetime, a dispatcher can acquire new memory by asking another entity in the system—what this entity is and how it interacts with said dispatcher is out of scope for this explanation. In the same way, it can return memory in case that memory is no longer needed.
When the parent wants to spawn a child, some of its own memory is formed into that of the child. In fact, it now still owns that memory, but rents that memory out to the child.

2.2.2. Loading The Binary

As a first step, the parent dispatcher has to check for the availability of the child's program code. The parent specifies a program name in the form of a file name.
Upon boot, the init dispatcher receives information about the statically compiled programs of the system. It can simply lookup the capability for the binary it wants to load, and map it into its virtual address space.

After that, it must check it the mapped memory is indeed an ELF. Otherwise, the newly spawned dispatcher would not be able to run.

2.2.3. Setting Up The Capability Space

The child process will have to make assumptions regarding its capability space. That is, it can rely on being provided with a certain amount of RAM, and being equipped with the root capability to manage its virtual address space.
First, we have to create a new capability of the type `ObjType_L1CNode`. This will be the root capability for its capability space. Later, when invoking the dispatcher, this capability is passed as an argument of the system call.
Then, three capabilities of type `ROOTCN_SLOT_SLOT_ALLOC0` are created on level two. This is necessary, so the child has slots available right from the start.
Next, we allocate a slot on level two, and completely fill it with RAM, each of size `BASE_PAGE_SIZE`. Listing 1 shows how this is efficiently done in our code using retyping.

```
struct capref cap_ram;

err = ram_alloc(&cap_ram, L2_CNODE_SLOTS * BASE_PAGE_SIZE);
if (err_is_fail(err)) {
    // Error handling...
}

err = cap_retype(cap_start, cap_ram, 0, ObjType_RAM, BASE_PAGE_SIZE,
↪  L2_CNODE_SLOTS);
if (err_is_fail(err)) {
    // Error handling...
}
```

Listing 1: Create initial memory for the new dispatcher.

Lastly, a slot for the capability of the level-0 page is required. It is of the type `ROOTCN_SLOT_PAGECN`, and needs to be passed to the paging state of the child.

2.2.4. Setting Up The Virtual Address Space

Given the slot for the level-0 page from the previous step, we can create the actual VNode in it. This in done via a call to `vnode_create()`, where we have to pass `ObjType_VNode_AARCH64_l0` as the type.
We will need the paging state of the child in the following steps. Luckily, at this point we have already enough information to initialize the paging state of the child.

2.2.5. Parsing The ELF

As we now have a working paging state, we can start filling the virtual address space of the child. The code for the program to run has already been loaded in form of an ELF.

In a call to `elf_load`, we can specify a callback to copy all segments into the address space.
The callback is not as trivial as it seems at the first glance. We are given a segment of the ELF, and need to return a pointer where the parsing library will copy the segment to. One thing to note is that we need to be very careful about the exact size we allocate for a segment. The passed base address and the size of the segment will not always represent valid boundaries for frames in our system.
The segments need to mapped into both the parent's address space and the child's. The parent needs to write the segments into the memory, while the child needs to read the code for its execution.

2.2.6. Setting Up The Arguments

As any program, our child will expect its arguments to be accessible from memory. The program name may contain arguments, which we have to parse and put into memory in a compatible fashion. To be more specific, we need to create an array containing pointers to the individual arguments. The arguments need to be zero-terminated.
The setup is done in what we call the arguments page. We require all arguments to fit into this single page. Note how the addresses used by the parent to setup the arguments are different from the addresses used by the child to read the arguments. Similarly as with the loading of the ELF segments, we hence need to map the memory into the parents paging state and into the child's one.
Listing 2 shows how we prepare the arguments page. Here, `params->argv` is the array passed to the main function of the child.

```
for (int i = 0; i < argc; i++) {
    strcpy(argv_data, argv[i]);
    int n = strlen(argv[i]) + 1;
    params->argv[i] = argv_data - (char *)args_page_parent + *args_page_child;
    argv_data += n;
}
```

Listing 2: Setting up the arguments for the child.

2.2.7. Setting Up The Dispatcher

The last big step is to initialize the actual structure that represents the new dispatcher as an object. Surely, this is a capability on its own, and we can create it by calling `dispatcher_create()`. However, this only gives us half of what is needed. Most information is stored inside a dispatcher frame—a normal frame of size `DISPATCHER_FRAME_SIZE`. The child expects to have access to

- its own dispatcher capability,
- an endpoint capability to the child itself, and

- an endpoint capability to the different services, namely serial server, process server, init server, and memory server.

Hence, these need to by copied into the child's capability space.
Similar to the arguments page, the dispatcher frame needs to be copied into both address spaces. The frame is then casted to a `dispatcher_handle_t`, from where we can write its entrypoint, the section address of the GOT, and the name of the process, among others.

2.2.8. Invoking The Dispatcher

Finally, we can call `invoke_dispatcher()`. This is where all puzzle pieces are processed, so the new dispatcher can run. The important bits that we pass here are

- the root capability of the child's capability space,
- the root capability of the child's address space, and
- the child's dispatcher frame.

2.3. Optimizations and Improvements

Verifying Intuition We argued our use of data structures in the paging system by intuition. It certainly makes sense to compare different approaches and to verify the performance by benchmarking different implementations.

Tracking Free Address Regions As described we use a doubly linked list. Data structures with a better complexity should give a speed improvement.

New Virtual Address When a user wants to map a frame without specifying a virtual address our implementation takes the first address with enough space and the right alignment. This can introduce external fragmentation. Due to the sheer amount of possible virtual addresses in ARMv8 the issue is negligible in our case. However, when targeting different architecture this should definitely be considered.

Add `unmap` We did not implement an `unmap` function. This would be needed for a commercial system. For our use case it is not necessary, as the uptime of our system will be rather low compared a ready for use system. The underlying data structures support removal of nodes, but the functionality was never integrated into the higher-level systems.

3. LMP Message Passing

> *Problems are often stated in vague terms... because it is quite uncertain what the problems really are.*
>
> — John von Neumann

Recall that we implemented dispatch of new domains in the last chapter. This milestone takes on the task to implement a way for these domains to communicate with each other. We make use the CPU driver's ability to send Lightweight Message Passing (LMP) messages.

3.1. LMP in Barrelfish

LMP is based on endpoint capabilities and the way how Barrelfish exchanges messages between domains on the same core. An LMP channel contains the receiver's dispatcher and refers to a buffer to store incoming messages. In order to send a message, the sender invokes the endpoint capability of the receiver. This instructs the CPU-Driver to transfer the message into the endpoint buffer of the receiver and makes the receiver domain runnable.[1] The CPU-Driver is nonblocking which is why LMP is non blocking as well. If the receive buffer is full, the operation fails and has to be retried. One could argue that a non blocking RPC mechanism is inefficient. In Barrelfish, LMP will always make the receiving dispatcher runnable and with a clever hint mechanism, a newly scheduled domain can quickly figure out what has been sent.
With these LMP facilities, we introduce an LMP Server framework which allows domains to provide and access RPC services. We then go on to detail how we designed such services and what functionalities they provide. At the end, we have intra-core RPCs which provide ram capabilities, spawn new domains, send messages to init and do serial I/O.

3.2. Implementing a Server Framework

In order to reduce code dublications we introduce an LMP server framework. All RPC services implement similar functionality. They all implement buffering incoming chunks, executing some business logic and formulating a response.

[1] Source: the book provided by the course and `http://www.barrelfish.org/publications/TN-011-IDC.pdf`

3.2.1. Establish a Connection

A crucial part of any communication is the initiation process, that is how do the parties setup a connection that can be used for communication. As mentioned before, for LMP we need capabilities to identify the endpoints of a communication channel. This means we need some way of exchanging these capabilities between dispatchers that want to communicate with each other. But how can we transfer those endpoint capabilities if we don't have any means of communication? If we did not have any means of communication this task would be impossible for obvious reasons! However, we do have one way to communicate, namely while spawning. That way we can pass an endpoint capability to a new dispatcher. This changes the question on how we are going to establish a communication channel. The new question is "how do we establish a communication between two parties if the endpoint of one is known?" We could think of the following two high-level server client concepts:

- Send an identifier which each query to the known endpoint which acts as a server. That way the endpoint knows to whom to respond to. In our setting, this would mean that we send an endpoint capability with each request.
- Perform a sort of handshake to create a dedicated endpoint for each client. In other words each client sends a newly created endpoint capability to the server, which will then in turn create an exclusive endpoint capability which is returned to the client.

The advantage of the first variant is that it is extremely simplistic and can be implemented quickly. However, this simplicity comes with its downsides. First of all, we would have to transfer an endpoint capability in each request, which implies a performance penalty. What's more, barrelfish only allows to send one capability per syscall. What if we want to transfer another capability? This is certainly an important use-case, as we want to transfer capabilities in later stages of the project. We considered some workarounds that we quickly rejected as they were way inferior to the second design. The second design bears only a tiny bit more complexity without the aforementioned downsides and is the design we chose to implement.

3.2.2. Open Channels and Service Channels

During initialization, our servers create something we call an *open channel*. This is an endpoint which is used in an open-receive fashion, i.e., multiple dispatchers can write to it. This endpoint is used to negotiate so-called *service channels*, which are then used to serve a single client's requests. The capability to the open channel is passed to new domains during spawn. Figure 3.1 depicts the communication establishment of a new service channel.

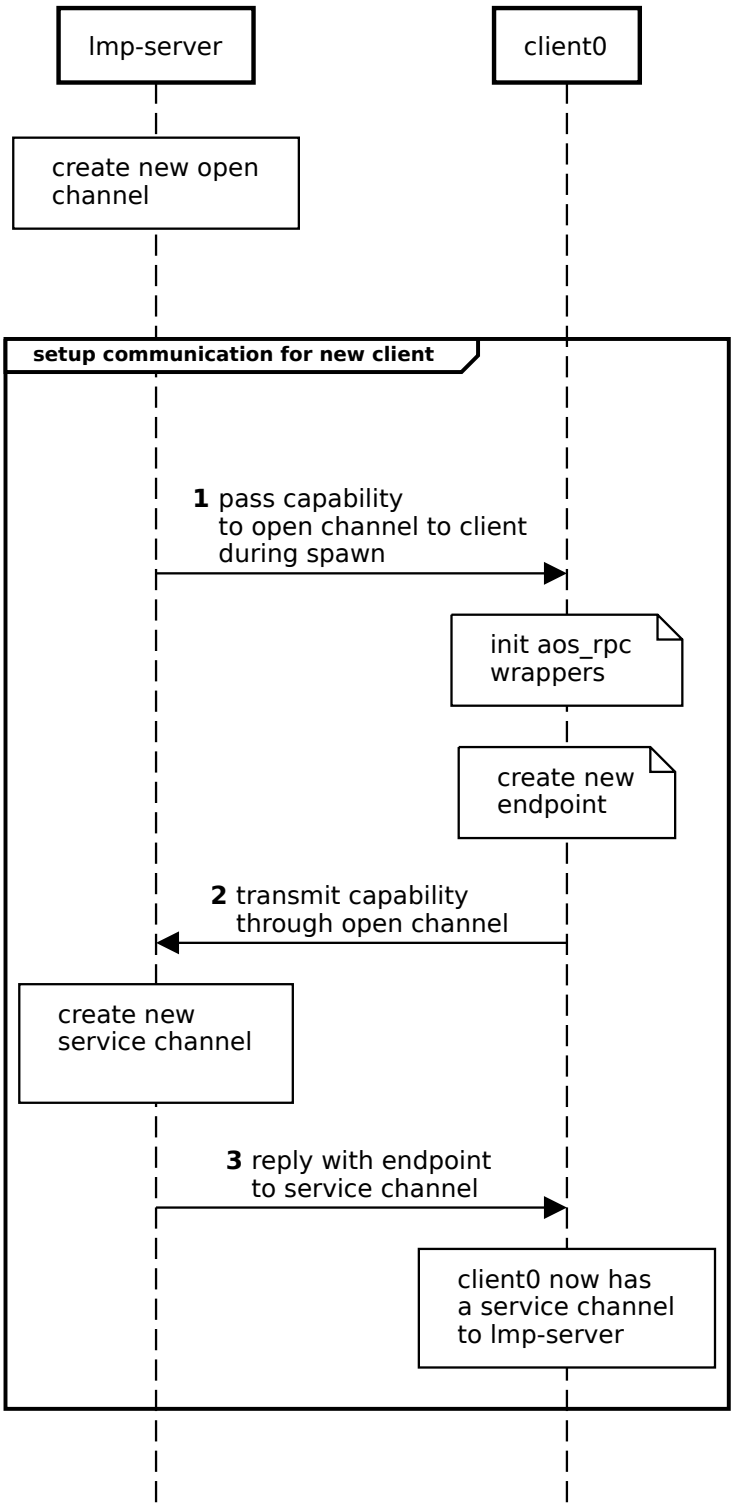

Figure 3.1.: High-level overview of open and revice channels

3.2.3. Abstracting General Server Routines

In total, there are four servers that we have to implement. All of them share similar behavior in regards to setting up connections and handling tasks. In order to make our servers less error prone, we first exported the relevant code into a separate module named `rpc_lmp_server.c`. All of our servers use this generalized server to implement their server functionality as shown in figure 3.2.

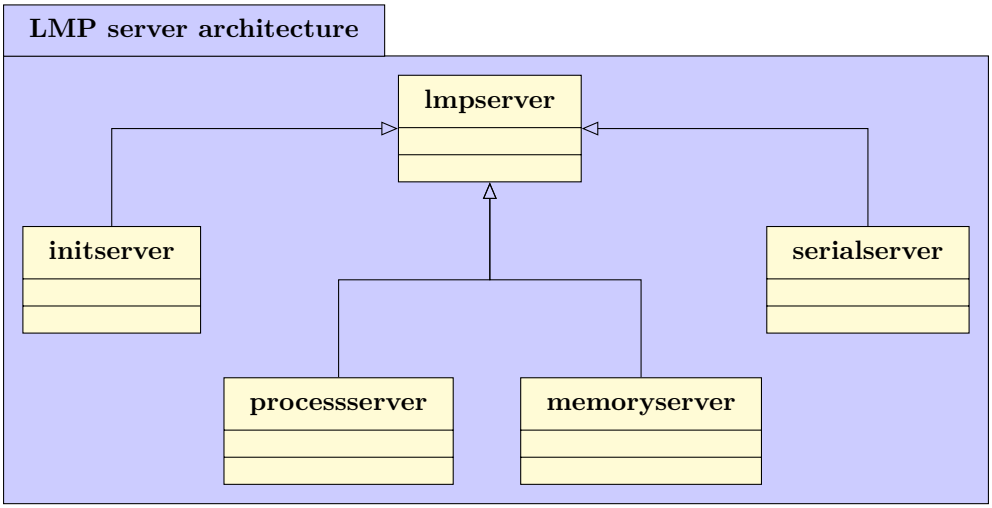

Figure 3.2.: The service servers "inherit" the functionality of the generalized server.

The generalized module takes care of two important tasks for the implementing server.

- It sets up the open channel where other channels can negotiate a dedicated channel for further communication. When a new request on this channel is received, the implementing server is notified via a callback.
- Incoming messages on the service channel will be marshalled into a single request message, and passed on to the implementing server. This means, the implementing server does not have to put the fragmented pieces together by itself.

3.2.4. Init Server

The `initserver` module implements the receiving side for the two required functions

- `aos_rpc_send_number()`, to send a number, and
- `aos_rpc_send_string()`, to send a string.

To do so, it makes use of the generalized server, and registers a callback for the service channel. Since the type of the message is encoded in the transmitted message, the `initserver` module can differentiate between said functions. This is the procedure for all other servers, except when stated differently.

Transferring Strings of variable Size

`aos_rpc_send_string()` accepts a string of variable length. This comes with some design challenges. In a single syscall, only a limited number of words can be written to the associated receive buffer of the target domain. We can either

1. send fixed sized strings only,
2. break a string into multiple messages,
3. map a shared frame into virtual address space of both domains and transfer string through that frame.

While limiting the payload to a single syscall is the most straightforward implementation, sending a string of only a few machine words is not very usable. On the other hand, breaking a string into multiple syscalls requires a concept of an entity; multiple syscall resemble a single string. In respect to other RPC services we will likely require transferring messages larger than a syscall in any case. Hence, an intermediate level API should implement buffering and assembling messages. This intermediate level API can either implement buffering as described, or map a shared frame. A shared frame will likely scale better in the size of the payload. However, setup and teardown logic is an additional engineering effort and it is reasonable to assume that we send small strings. Given this reasing, we opted for the second design. This abstraction is further described in section 3.3.

3.2.5. Process Server

The `processserver` module implements the receiving side for the three required functions

- `aos_rpc_process_spawn()`, to spawn a new dispatcher,
- `aos_rpc_process_get_name()`, to retrieve the name of a dispatcher based on its PID, and
- `aos_rpc_process_get_all_pids()`, to retrieve the PIDs of all running dispatchers on the system.

To generate consistent replies to the requests, the process server needs to maintain state. When a new dispatcher is spawned over LMP, it is registered at the process server, where it is added to a linked list. All nodes in this list have a name in form of a string and a PID associated with them.
For looking up the name of a dispatcher, we need to traverse this list until we find a matching PID. Retrieving the PIDs of all running dispatchers is more convenient—we can simply traverse the list, and add all PIDs to our response.

3.2.6. Memory Server

The `memoryserver` module implements the receiving side for the required function `aos_rpc_get_ram_cap()` that is used for requesting more memory.
The memory server is a critical part of our infrastructure. Other dispatchers are highly dependant on this component, as they do not have other methods to obtain more memory. For this milestone, we wanted to come up with a stable design, without making too many fundamental changes. For instance, this means that we agreed that for now, the memory server can continue running on the initial dispatcher.
Why does this make life easier for us? If we had a separate dispatcher acting as the memory server, the initial dispatcher would first have to carry out the memory server's work on its own. After the memory server was spawned, it would have to switch over to using it itself. So far, so good. But the problem lies in the detail. While the initial dispatcher is running, it consumes RAM capabilities. When the dispatcher for the memory server is started, it somehow needs to be made aware of which parts are still free. This could be done in various ways, but we restrained from implementing this at this point.
In essence, what the memory server does is exposing libmm over RPC calls. Since there is only one type of command, the receive callback is fairly straightforward.

3.2.7. Serial Server

The `serialserver` module implements the receiving side for the required functions

- `aos_rpc_serial_putchar()`, to print a character to the console, and
- `aos_rpc_serial_getchar()`, to receive a character from the standard input.

The basis for this server was already implemented in the first milestone, where the serial output had to be written. The serial server now offers writing characters to that output stream. This way, only a single, central entity is in charge of writing output to the console.
Similarly, the reverse direction is exposed. Dispatchers can request to receive a character from the serial input. We implemented this in a simple manner, which comes with a limitation. Since the user does not necessarily write something on their keyboard as soon as the dispatcher requested a character, the server must wait. In our case, since all servers currently run on the same dispatcher this would also block the memory server, and with it, all other dispatchers that request memory. A more useful implementation will have to buffer incoming requests and correctly multiplex them to who currently is reading. In a later milestone this will be implemented in greater detail.

3.3. Channel Abstractions

Dealing with the raw syscall interface of LMP is rather tedious. This is why we provide an intermediate level API to send arbitrary content between dispatchers. A message

is abstracted in `struct rpc_message`. With a `payload_length` we indicate how many bytes are contained in the payload and a zero-length array [2] as the last element of the struct holds continuous memory for the payload. With a `status` and a `method` value we can transmit errors and multiplex messages. There is an enumeration value for all RPC function (PutChar, GetChar, Spawn and so on). Our framework provides functions to fire and forget a message and to send and block until a response arrives. This design is given by the required functionalities to implement. Under the hood we do a `SYSCALL_INVOKE` and invoke the receiver endpoint capability. We yield our timeslice to the receiver such that the recipient can process the request. While LMP is non-blocking at the level of the CPU Driver, we can block in userspace by waiting on an associated waitset until an event occurs. In a single syscall we can only transmit a limit number of bytes which is why we break down `struct rpc_message` into smaller chunks. This for instance is needed to transmit strings. Payload chunks are buffed on the receiving side. With these intermediate level primitives we implemented the RPC functionality of this milestone.

```
struct rpc_message {
    struct capref cap;
    struct rpc_message_part msg;
};

struct rpc_message_part {
    uint32_t payload_length;
    uint16_t status;
    uint8_t method;
    char payload[0];
} __attribute__((packed));
```

Listing 3: Datastructure built for the intermediate level API we expose to build RPC services. rpc_message_part contains a payload to transmit arbitrary data. Our primitives further allow to transmit capabilities with LMP

3.3.1. Implementation

Implementing the intermediate level API was straightforward. We initially did not build a function to send and block until we receive an answer but rather implemented this functionality for all LMP calls ourselves. Namely this involved registering a callback on an endpoint and buffering incoming chunks of `struct rpc_message` until everything was received. This caused a lot of code duplication which is why we provided a separate function for it. It proved to be good practise not to spend too much time refactoring in the first iteration of the implementation. Instead we first ensured that we can send and receive LMP messages and later on refactored according to common software practises.

[2] flexible array member in 1999 ISO standard

Implementing marshalling stubs is tedious work and is best described by a DSL and then generated by a code generator. In fact, this is how it is done in mainline Barrelfish with Flounder.[3]

Identify messages with a Tag

In discussions how to design the intermediate API we considered introducing a tag in `rpc_message`. A tag had the function to identify chunks which belong to the same message. However, we realized that this is not needed because we establish an LMP channel for each domain and server (service channel). So two domains cannot interfere each others by both sending messages to the same rpc service.

3.4. Optimizations and Improvements

Resource Cleanup At this point, processes cannot hand back memory to the memory server. There was no interface specified in the assignment, and we felt as if that task would be conceptually straightforward. This means, a free will only locally free a certain memory region, so it can be reused by the very same dispatcher. To enable the actual freeing of memory, we would need to expose `mm_free()` via RPC calls.

Non-blocking Serial Input In a production system, we cannot have a serial server that blocks and waits until something is received on the serial connection. Here, we only showed how keyboard input could work, but we did not get into the details of it. If we had to improve it, we could spawn a thread that is dedicated to reading the input from the serial connection. This way, all the other servers (plus the serial output), would continue to work normally.

Outsource Service Servers Into Dedicated Dispatchers For practical reasons, all implemented RPC services run within init. This comes with the disadvantage that if init crashes, all RPCs within it crash. There is further no semantic connection between the services. They are unrelated with each other. For better fail safely, our RPCs should run in their own dispatchers. New domains need to know endpoint capabilities to all dedicated dispatchers. This can quickly become tedious and is best provided by something like a name server.

[3]Source: http://www.barrelfish.org/publications/TN-000-Overview.pdf

4. Page Fault Handling

> *A computer once beat me at chess, but it was no match for me at kick boxing.*
>
> — Emo Philips

In this milestone we implemented an exception handler that supports self-paging and in particular lazy memory allocation. Firstly, we start by describing our initial exception handler and how it interacts with the rest of the paging code. Secondly, we explore how we rewrote big chunks of the paging code in order to simplify the implementation of an `unmap` function. Lastly, we delve into our design decisions for the dynamic heap allocation.

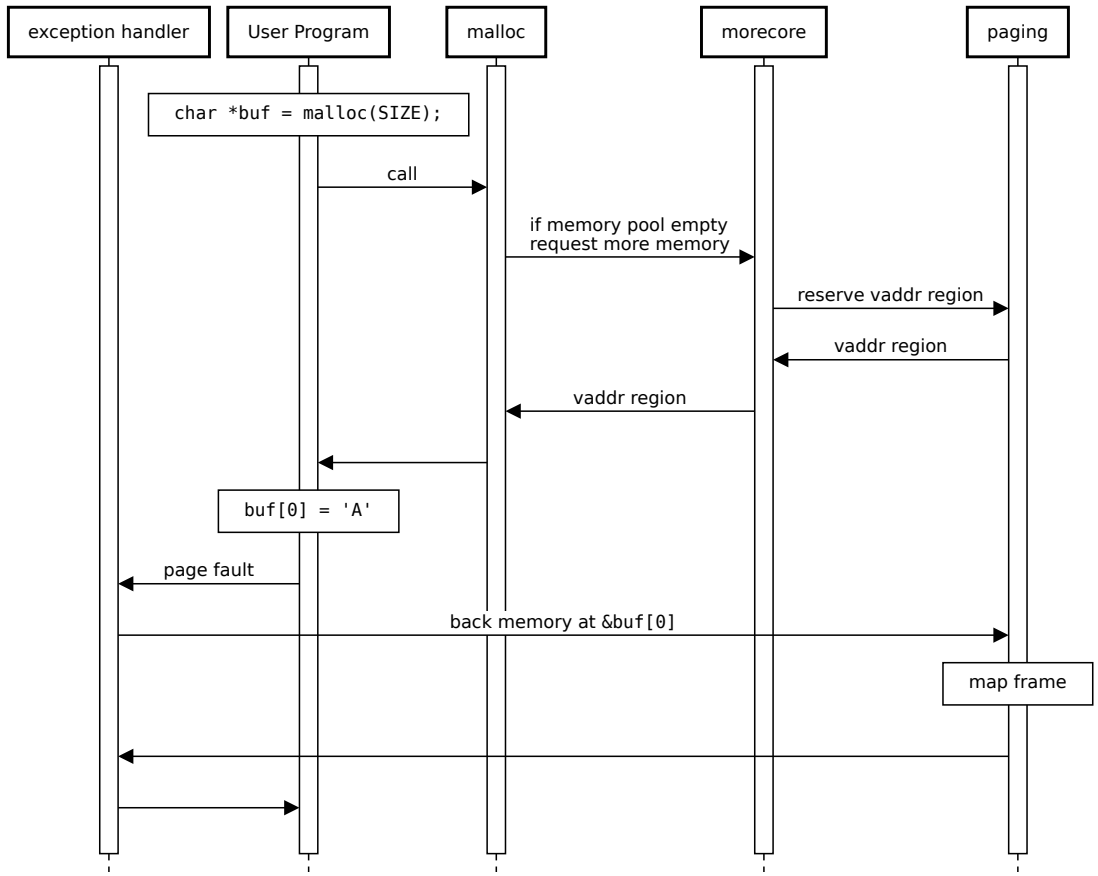

Figure 4.1.: High-level overview of lazy allocation.

4.1. Exception Handler

Setting up the exception handler is quite straight-forward. There were no major decisions to make. The relevant parts are found in `paging_init()` and `paging_init_onthread()`. For exception handling to work, we need some memory for the stack of the exception handler. Extending the paging state to provide this memory seemed like a reasonable thing to do. This way we can rely on non-paging functionality for the allocation.
The code being run when an exception occurs needs to carefully *handle* the exception. To do so, it receives information on what type of exception was raised, and which address is affected.
Our handler can only deal with a single type of exception at this time, namely with pagefaults. A pagefault is an exception where the page of the affected address is not mapped in the page table. The specifics of handling page faults will be discussed later in this chapter.
A call to `thread_set_exception_handler()` registers the handler together with its stack for the active thread.

4.2. Initial Address Space Representation

We already implemented basic functionality to track the used virtual addresses in section 2.1.2. In a first step, we simply extend the tracking with an additional state, such that virtual address regions can be marked as reserved. This allows us to know which addresses should be mapped when a page fault occurs. As mentioned in section 2.1.2, the addresses are kept as linked list. With the work previously done, we can quickly proceed with the page fault handler. However, due to some limitations in the previous implementation and some technical dept, we decided that some part of the team should rewrite parts of the paging code and the address regions tracking.

4.3. Refactoring the Paging System

One flaw in our previous implementation of the paging system was its unclean separation of the reserved state and the mapped state of a page. We realized that this should be done cleanly, so that dynamic memory regions can be properly handled. At this point we realized that there is not just a single state per memory area, but two.
Chapter 3 showed us the limitation in performance of our previous approach, so that another flaw of the initial design became clear.
To address these problems, we saw the necessity of a fundamental change in how the system keeps track of allocated areas.

4.3.1. Tracking Memory Regions

Our idea was as follows. The new system has to maintain some state that reflects the current allocations of the memory space. This state is separated in two layers.

1. The *upper* layer represents free paging regions, i.e., regions that are neither reserved nor mapped.
2. The *inner* layer maintains the mappings inside a paging region.

To represent this structure in memory, we use lists. In fact, the state is now a nested list. For the upper layer, each list node represents a paging region. For the lower layer, each list node represents a mapping into virtual memory. An illustration of this scheme can be seen in figure 4.2.

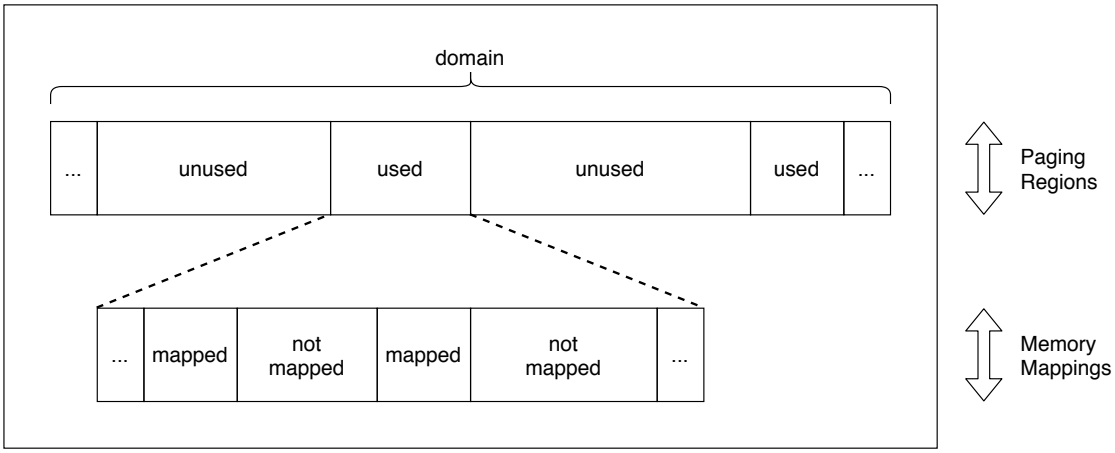

Figure 4.2.: Two-layer data structure for reflecting memory mappings.

Note how all areas that are not free are associated with a paging region. That is, even for direct mappings, which will not be lazily loaded by a page fault, we create an *implicit* paging region. The user of the paging system does not know about this, which is why we call it "implicit". This way, when we look for a free area, we iterate over paging regions, and can thus skip them as a whole. In contrast, when a user requests a paging region—which will then be destined for being lazily loaded—an *explicit* paging region is created.
The lower layer quite literally represents mappings in the page table. Since we can span a mapping over a whole L3 table, our implementation tries to make use of that where possible. As an implication, a page fault will map memory spanning over a whole L3 table, which equals 2 MiB.
Of course, when a paging region is created, a new node is created in the upper layer. This means with a growing number of mappings, the system will slow down linearly. Still, the factor is reduced due to the second layer we introduced.

4.4. Generalizing Memory Range Tracking

As is the case for the memory management in libmm, we have to keep track of memory regions in the paging system. In fact, the paging system also keeps track of mappings within a paging region, so that the same pattern now occurrs three times in our code!

At this point, we felt the urge to introduce a module that abstracts this task away. The result was a data structure called *range tracker*. A range tracker is given an initial "range" that it operates on, a so-called *domain*. This is done in the sense of base blocks, which we have already seen in libmm. The structure then provides functions to allocate different sizes in its domain.
Conveniently, the caller can associate data with allocated blocks. Optionally, we can request the range tracker to allocate a block at a specific location. It handles both allocation and freeing, as well as checking the state of a certain region in its domain.
As a side note, the range tracker itself can be properly tested more easily. We have found that testing paging code is especially difficult, if the test itself cannot rely on the underlying memory mechanics. For us, this was a great relieve, since a lot of complexity was moved from libmm and the paging system into a dedicated module, which on top could be easily tested.

4.4.1. Merging

Under the hood, a range tracker is based on a list. When allocating a new block, it will start iterating from the head of the list until a match is found. When freeing a block, the corresponding list node is marked as free, and it is fused with neighboring free blocks. This ensures that the list size is kept at a minimal possible size, and allocations are most efficient. An illustration of this can be seen in figure 4.3.

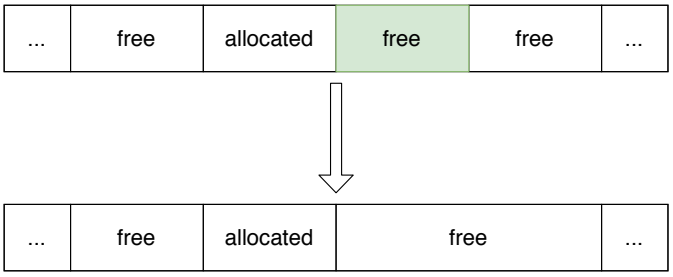

Figure 4.3.: When a block is freed, it needs to be merged with free neighbors.

4.4.2. Internals

The range tracker module is very flexible and generalized, and supports closures for tasks like freeing of ranges. In essence, there are four types of methods.

Initialization

During initialization, the underlying data structure is set up.
Since the range tracker is to be used for memory-critical tasks, it takes a slab allocator, with which new nodes of the list are allocated. Hence, no dynamic memory allocation other than that using the slab allocator is performed. This is why we can use the range

tracker for managing memory in libmm. On a side note, libmm needs to supply its custom slab allocator, which luckily is compatible with the slab allocator interface.
At initialization, the user can also decide to supply an alignment for the region boundaries. This makes it very powerful for our memory regions, because we often need to adhere to given alignments, too!

Adding Base Blocks

Before allocating blocks, the domain has to be specified. This is done by adding n base blocks. Each base block b_i with $0 \leq i < n$ is defined by its base and a size. Then, the domain D of the range tracker is

$$D = \bigcup_{i=0}^{n-1} b_i.$$

Even though multiple base blocks are supported, our current architecture only makes use of a single one. In case of the paging system, we want to operate on addresses between `0` and `VADDR_OFFSET`. Thus, we need to initialized the range tracker with a base block matching these parameters.
In libmm, we face the additional challenge that we need to associate a RAM capability of the major memory block with the ranges of the range tracker. This is why we equipped the list nodes in the range tracker with so-called shared objects, which represent associated data of the node. This data is to be specified by the user of the range tracker, and can be either a void pointer or a reference to a capability. To maintain compatibility with the current behavior of the range tracker, the internal functions will never merge two nodes with different shared objects.

Allocating Blocks

For allocating a block, there are two options. Either, we can specify a base and a size, and the range tracker would try to allocate that very specific region for us. If the region is already allocated, the region tracker would return an error.
What can be done instead, is to specify just a size, and the range tracker is free to choose where that region is located. As long as some contiguous region of this size is available in its domain, the range tracker would be able to successfully allocate a block.

Freeing Blocks

To free blocks, the user again specifies a base and a size. Instead of using a handle, we decided that this way, the user does not have to adopt a new concept for the use of the range tracker.

Finding Blocks

Lastly, the range tracker can be instrumented to find a certain range in the domain. For instance, the paging system might be interested in whether or not a specific paging region is implicit or explicit.

4.5. Implementing Paging Regions

With this milestone we introduced a proper concept of paging regions. This means that functions regarding paging regions, e.g., `paging_region_map` were implemented.
Paging regions are originally thought of as contiguous regions in memory for a specific purpose. In their meaning derived from the assignment, they should be lazily loaded when they are accessed.
In our system, we use the term "paging region" for a more general concept: each address belongs to a paging region. This does not necessarily mean that all addresses are lazily loaded. For instance, if a dispatcher was to map a frame directly, the paging system would register it as an implicit paging region.
Exactly this is the purpose of the `paging_region_*` functions in our architecture: they operate on explicit regions.
A minor detail to note is that when an explicit paging region is created, the corresponding range tracker will only contain a single node. In other words, the splitting of the regions inside the paging region happens lazily in the time of use.

4.6. Page Fault Handling

At first, we implemented a simple page fault handler based on our original memory tracking system introduced in section 2.1.2. That way we could test and start implementing the dynamic heap, which is described in the next section. It was a strategic decision to spilt the work into two parts. In this section, however, we will describe the page fault handler using the new range tracker.

4.6.1. Illegal Page Faults

Some page faults cannot be handled by the handler, simply because the requested address is outside of its domain. Our page fault handler tries to detect these and act accordingly.

1. If the address is `NULL`, then we assume that a `NULL`-pointer dereference occurred. Conventionally, `NULL` is used as a special address, indicating that a pointer references no valid data.

2. Similarly, if the address belongs to the first base page of the virtual address space, the page fault handler cannot serve the request. This is because in fact, the whole first base page will stay unmapped throughout the whole run of a program.

3. Lastly, the address might belong to a different paging state. In this case, the page that this address belongs to is not managed by the current paging state.

   To explain how this can happen, let us briefly consider how a dispatcher is setup by its parent. Since certain data needs to be passed from the parent to the child, the parent needs to initialize a paging state. This paging state operates on a predefined zone of the virtual address space. After spawning the child, that predefined zone

was already operated on, so the child cannot know about any mappings within that zone. Hence, we disallow the child's paging state to manage this zone.

4.6.2. Paging Region Types Revisited

In case we handle a legal page fault, this means the page fault could be handled by mapping a frame into the corresponding page table entry. But this is not always the correct thing to do.
Recall how we are using explicit and implicit paging regions. For the latter, no dynamic loading should be needed. This means before mapping the frame, we first have to check for the type of paging region that the specified address belongs to. Only *explicit* paging regions will be handled by mapping memory at the given address; *implicit* ones will result in an unrecoverable error.

4.7. Dynamic Heap Allocation

In order to support lazy mapping, `morecore`—which is the equivalent to Linux' `sbrk` syscall—should return unmapped but reserved memory regions. The reserved regions should be mapped when a page fault happens. In theory this sounds simple and straightforward, however there are several memory dependencies we introduced along the way that we have to keep in mind. A simplified dependency graph is shown in figure 4.4, if we apply the mentioned idea to our current design. One can immediately see that we have several loops. The loops related to libmm, the slot allocator and slab allocator have been resolved in chapter 1. The interesting loop is involving morecore, the paging system and `malloc`.

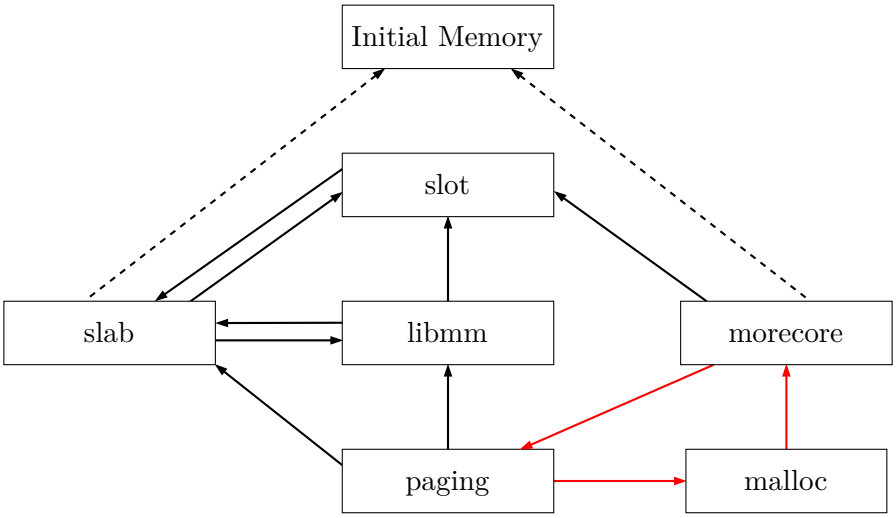

Figure 4.4.: Memory dependency graph

Since our paging code needs dynamic memory allocation, we depend on memory from `malloc` and thus on memory from morecore. Prior to this chapter, morecore took all its memory from a static buffer, which is denoted as "Initial Memory" in the graphic. Therefore, the dependency from morecore to paging did not exist. However, as we aim to support lazy allocation, we must introduce such a dependency, which comes with its problems. Consider the scenario, where a page fault happens to map a frame lazily. A `malloc` call in the paging code can create another page fault, but we are already in the page fault handler... After some brainstorming, we came up with the following three possible solutions.

Use slabs One approach we considered is to rewrite the paging code to be independent from malloc. We already briefly considered this approach in our first paging implementation in section 2.1, and made some attempts in rewriting the provided hash table implementation. However at that time we scratched the idea quickly, because we needed memory allocations of different sizes. Provided we don't want to waste too much memory, we will need to use a different slab for each size with this approach. And using multiple slabs makes the code quite messy.

Adapt malloc Another way to go is to modify `malloc` to have two pools. One with backed memory which is supposed to be used by the paging code and a pool of potentially backed memory (i.e. memory that is only backed when it was accessed at some point) that might trigger a page fault. The advantage of this method is that we don't have to rewrite the paging code and can use the simple interface to `malloc`. A question that arises is how to change which memory pool to use without changing the interface to `malloc` as this is standardized.

Static heap for paging The last method we could think of was using an independent heap in the paging code built from a static buffer, i.e. the initial memory. While this approach is quite simple it is more an ugly hack in this form than a solution. The size of the static buffer is limited and it is not really future safe. Of course one could add a feature to refill the heap with frames when it runs low on memory, but then what is the advantage over adapting `malloc`? At this point we figured the sole advantage would be a cleaner abstraction and less entangling between different components. However, we would rewrite similar code to `malloc`, i.e. having a memory pool that needs to be refilled when running out of memory.

We eventually decided on adapting `malloc`. The decision was motivated by the fact that we felt like the other options were not appropriate in our situation. We already had a good feeling of what it means to replace the `malloc` calls with slabs in the paging code. We were sure that this results in messy code. Something we wanted to avoid at all cost. It would make debugging more painful and it is not a clean design. Using another heap was in our opinion only acceptable if we provided a refill function. That way it is probably the cleanest solution, as it is another clean abstraction. However, we wanted to safe the time of rewriting such a heap that is similar to `malloc` and rather spend the time writing more tests and gain confidence in our implementation. We thought this

approach is a better investment in the future of the project. Moreover it provides a nice trade-off between code complexity and estimated time to implement.
An overview of the dependency graph using the adapted `malloc` is depicted in figure 4.5. Morecore should be able to provide backed memory or reserved memory regions. We

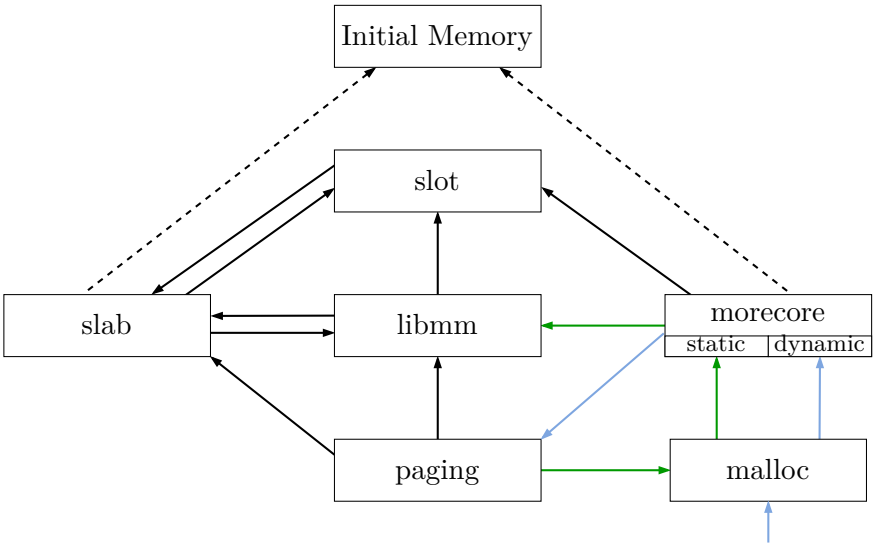

Figure 4.5.: Breaking the loop

denoted this with the rather confusing names static and dynamic. The former being backed memory, the latter the reserved regions that are only mapped when a page fault occurs. The origin of the confusing naming and how the individual morecore "backends" work is introduced shortly. First we explain how we handle the change of modes without changing the `malloc` interface.

4.7.1. Swapping between Static Heap and Dynamic Heap

As changing the signature of `malloc` and friends is not an option, we decided to introduce some global state for morecore and `malloc`. Put differently, we have to extend the morecore state with a boolean flag, that says whether we are in the static state, i.e. backed memory should be returned `morecore_alloc`, or dynamic state. The following invariant must hold: `morecore_alloc` returns backed memory if and only if it is in the static mode. Naturally, it is also essential that the `malloc` implementation is aware of this split, otherwise the memory would land in the same memory pool of `malloc` and the split was useless. That means we have a similar invariant as for morecore. `malloc` must return backed memory if and only if we are in the static mode. The provided `malloc` of barrelfish allows to easily replace the malloc function. Our malloc implementation is in essence a copy of barrelfish's K&R implementation. At this point the only change we performed, was introducing two magic numbers. One for the static pool and one for the dynamic pool. The idea behind this design decision is that a `free` call should

free memory into the correct pool, even if the global morecore state (which also affects malloc) does not correspond to the state when the to freed memory was allocated.

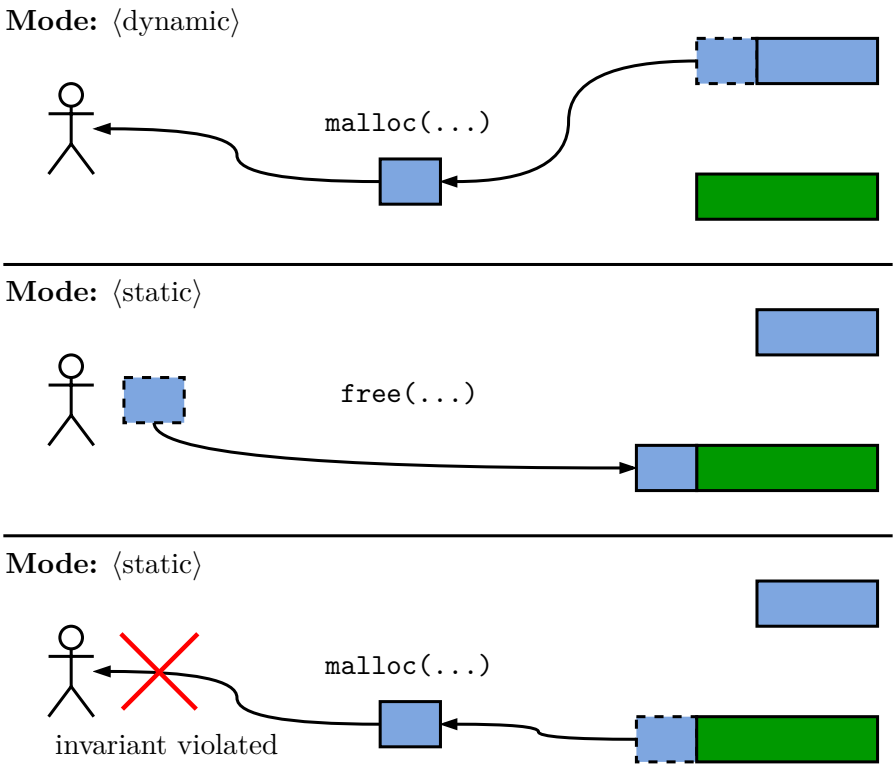

Figure 4.6.: Freeing breaks invariant without 2 magics.

If this was not the case, the invariant wouldn't be respected as one can see in figure 4.6. A question remains, how do we swap between the pools in the `malloc` call? As already mentioned our `malloc` implementation is almost a complete copy of barrelfish's one. We didn't want to change the implementation too much, as a bug there would be devastating and expensive to debug. That is why we opted for a design where a call would swap the pointers to the data structure used internally by `malloc` and friends. By calling `morecore_enable_static` the state of `morecore` and `malloc` is set to static mode. `morecore_enable_dynamic` can be used to go back to the version that returns unbacked memory regions. To resolve any ambiguity, we included `morecore_enable_static` in listing 4. `state->header_freep` and `state->header_base` are the variables used by `malloc` to keep track of allocated and free memory.

4.7.2. Static heap

The name static heap comes from the fact that originally a static heap with a fixed size was used in morecore. In our design we start with a static buffer with the size of 1000× `BASE_PAGE_SIZE`. This initial memory is used to bootstrap dynamic memory.

```
void morecore_enable_static(void) {
    struct morecore_state *state = get_morecore_state();
    if (!state->heap_static) {
        state->heap_static = true;
        state->header_freep_dynamic = state->header_freep;
        state->header_base_dynamic = state->header_base;
        state->header_freep = state->header_freep_static;
        state->header_base = state->header_base_static;
    }
}
```

Listing 4: Enable the static heap for `morecore` and `malloc`

This is required as `malloc` is used while setting up `init`. Additionally, while initializing morecore, we call `paging_alloc` to reserve a memory region which we will use to map frames eagerly, when a certain threshold of available memory is reached.

> Note: From the description in the book, this is probably not how you expect `paging_alloc` to work. `paging_alloc` is called to reserve virtual address regions that you are going to map yourselves with frames. Memory in this region will NOT be mapped when a page fault occurs. To achieve automatic mapping in case of a page fault, one has to reserve a region with `paging_region_init_aligned` and then `paging_region_map`.

The threshold is needed to perform the rpc calls required for `frame_alloc`. The rpc calls will use `malloc` to request RAM from the memory server to create a frame. As we are in the static mode and we will request once again memory from morecore. Due to the threshold those requests can still be served. As mentioned, the morecore module knows based on current state whether it should provide backed memory as described here or only a memory region which will be backed lazily.

4.7.3. Dynamic heap

For the dynamic heap we reserve a large paging region of 1 TiB. Due to the large address space in ARMv8 this is possible without any issues. Once `morecore_alloc` is called in the dynamic state, `paging_region_map` is used to tell our system that the addresses returned by `morecore_alloc` should be mapped on page fault. This separation between reserving addresses and specifying them as lazily mappable comes from the design of the range tracker and also allows greater flexibility as one might reserve regions for other purposes.

4.7.4. Interaction with paging module

To sum up our design, we reiterate on the important aspects of our design with respect to the paging code. We need dynamic memory which should not cause a page fault when written to, as we might already be in a page fault handler. Even if we are not, we would

go into the page fault handler, when `malloc` would return not backed memory. And this time we would be stuck at another `malloc` call. Another reason that also requires that no page fault happens in the paging code is the consistency of our data structure. A page fault could happen in an inconsistent state of the paging data structure and the page fault handler will change the data structure as well. This might result in a corrupt state of the shadow page table. To avoid those issues we designed our system such that it results in a memory dependency as in figure 4.5. As soon as we enter paging relevant code the heap is set to static mode. That way it is guaranteed that no page fault happens in the paging code or while the duration of a page fault.

4.8. Optimizations and Improvements

Add other exceptions One could consider other exceptions that could occur and define default behaviour to take in these cases. A few exceptions one might consider: floating point exception, divide by zero, illegal instruction.

Decoupling `malloc` and paging The use of `malloc` in our paging code introduced a dependency that together with lazy allocation resulted in a symbiosis in our system. Under our circumstances at that time this was a sensible approach. It allowed us to focus on improving other parts of our code and writing more tests. However, in later chapter we will see that this approach caused quite some trouble in debugging concurrency issues. We believe a better abstraction between the two components can result in better performance and a easier debuggable system. The performance improvement would stem from less management overhead in morecore.

5. Multicore

Science is what we understand well enough to explain to a computer; art is everything else.

— Donald E. Knuth

Until this milestone, our fork of Barrelfish runs a single CPU driver and thus supporting a single core out of four ARM Cortex-A35 cores on the Toradex board. Spawning an additional core consists of the following major steps.

1. Use the already running CPU driver to instruct the System Control Unit (SCU) to bring up a second core.
2. Engineer a communication protocol for the two cores to communicate.
3. Manage available memory between cores.

The next sections describe design decisions and challenges when implementing multicore support.

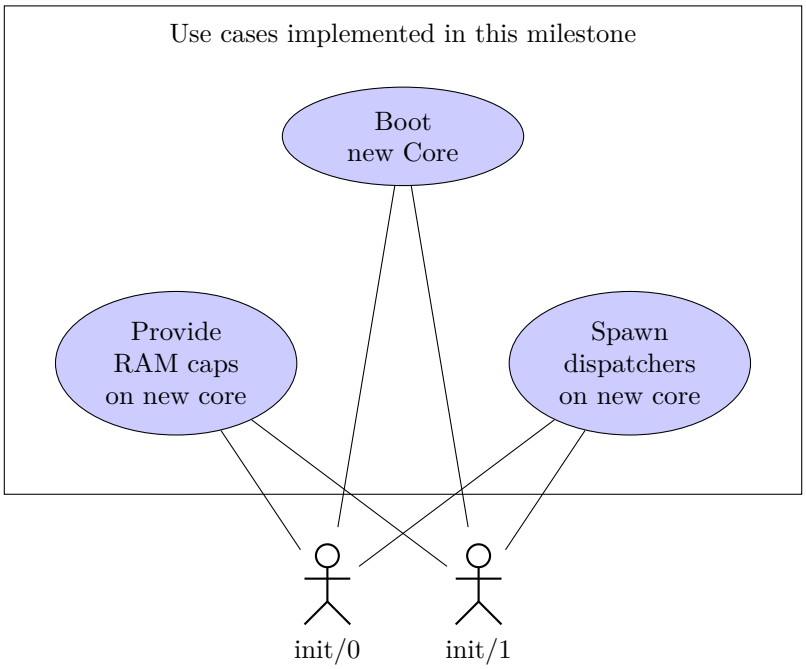

Figure 5.1.: Use cases. Actors are the init domains running on core 0 and core 1

5.1. Hello World on Core 1

Details on how to boot a new core are architecture specific. On Barrelfish, each core runs an own lightweight exo-kernel (CPU driver) and thus enabling the OS to run on heterogeneous hardware. Platform specific instructions how to boot core 1 on the Toradex board where given in the handout. This part of the milestone proved to be straightforward and implemented within an afternoon.
On a high level, it requires allocations of memory for a new CPU driver, init dispatcher, and further data structures. Loading the Boot and CPU driver into memory, flushing caches and invoking kernel capabilities to start the boot driver on the new core.

```
Barrelfish CPU driver starting on ARMv8 (APP) mpid=0:0:0:1
kernel 1: ARMv8-A: Global data at 0xffff000080200000
kernel 1: ARMv8-A: Kernel stack at 0xffff000080b7a000.. 0xffff000080b8a000
kernel 1: ARMv8-A: Kernel first byte at 0xffff000080b30000
kernel 1: ARMv8-A: Exception vectors (VBAR_EL1): 0xffff000080b30800
kernel 1: GICv3: Enabling CPU interface
kernel 1: GICv3: CPU interface enabled
kernel 1: ARMv8-A: Enabling timers
kernel 1: isr_el1=0x0
System counter frequency is 8000000Hz.
Timeslice interrupt every 640000 ticks (80ms).
kernel 1: ARMv8-A: Setting coreboot spawn handler
kernel 1: ARMv8-A: Calling arm_kernel_startup
kernel 1: ARMv8-A: Doing non-BSP related bootup
kernel 1: ARMv8-A: Memory: 80b8c000, 814f5000, size=9636 kB
kernel 1: ARMv8-A: spawning 'armv8/sbin/init' on APP core
kernel 1: ARMv8-A: spawn_init_common armv8/sbin/init
```

Listing 5: Excerpt of CPU driver first time booting on core 1

At this point, we have a second core running the init domain `arv8/sbin/init`. Conceptually as well as technically, there are some differences between the bootstrapping processes of core 0 and core 1. The init domain on core 1 boots into an environment where all RAM capabilities have already been claimed by the memory manager on core 0. The cores need a way to communicate with each other in userspace to manage memory along other userspace functionalities.

5.2. Communication between Cores

We keep the initial implementation of inter-core communication simple, given the fact that user level message passing is introduced in the next chapter to a more detailed extent.

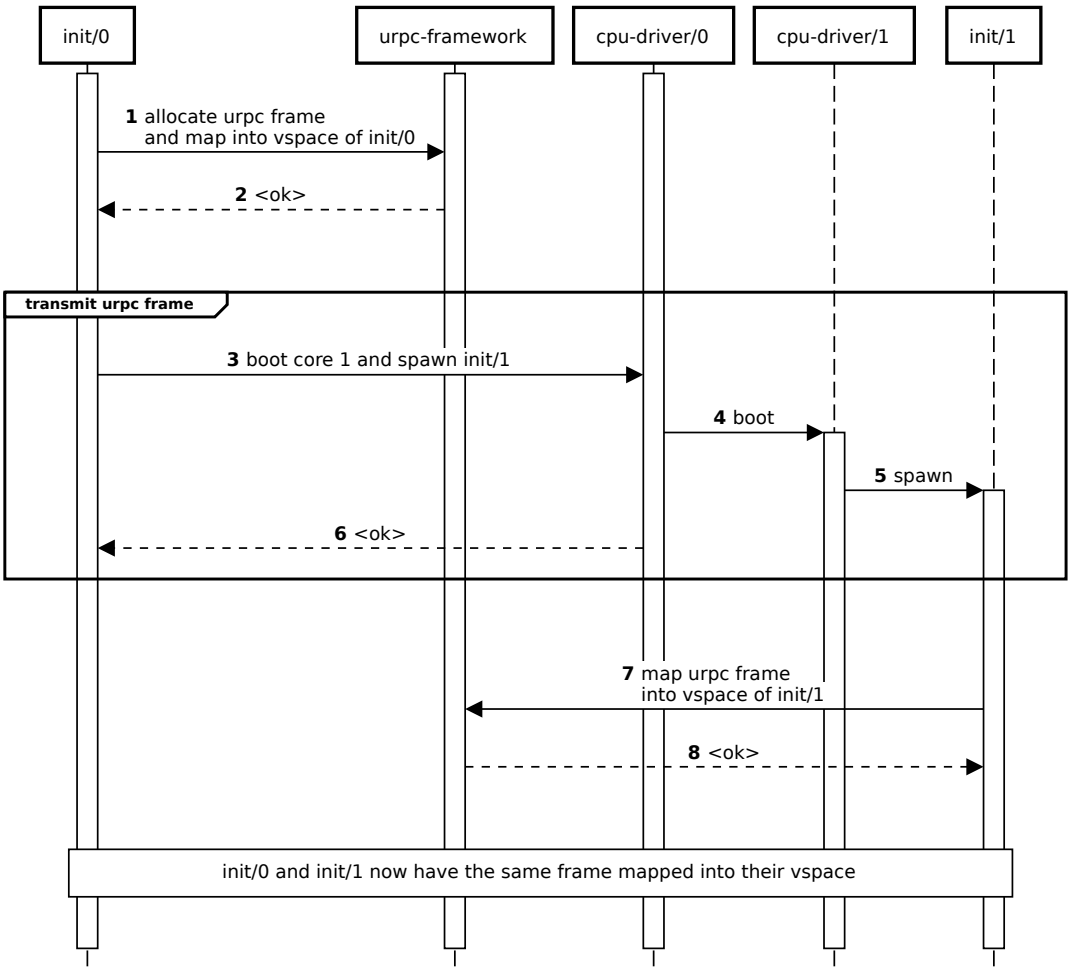

Figure 5.2.: Boostrapping of Core 1 and Transmission of URPC frame

The communication protocol relies on polling of memory in a shared frame, subsequently referred as URPC (User level RPC) frame. The capability to the URPC frame is exchanged in bootstrapping of core 1. A `urpc-framework` implemented in this milestone provides functions to transmit data between the two cores.
The sequence diagram in figure 5.2 shows bootstrapping of `cpu-driver/1` (`/1` refers to the core number) and spawning of `init/1`. At the end of bootstrapping, we have the same frame mapped into vspace of `init/0` and `init/1`.

5.2.1. URPC Framework

Our userspace framework relies on a master/slave architecture where `init/0` is in master role and `init/1` is slave. This approach is suggested in the handout. In order to keep the initial version simple, we further use a single chunk of memory in the URPC frame. Either master or slave can send a single message at the time and a subsequent message

can be sent if the other party replied. The exchange of message starts by the slave pulling for messages. Figure 5.3 shows a message exchange of the first message on the URPC channel. This design is simple and wastes throughput. The URPC framework acts as a serializer. Polling on the shared memory location is done in a separated thread. Alternatively it can be implemented in an event driven way with waitsets but threads seemed more straight forward. A `status` indicates if the chunk is for master or slave and a `type` describes the layout of data to exchange.

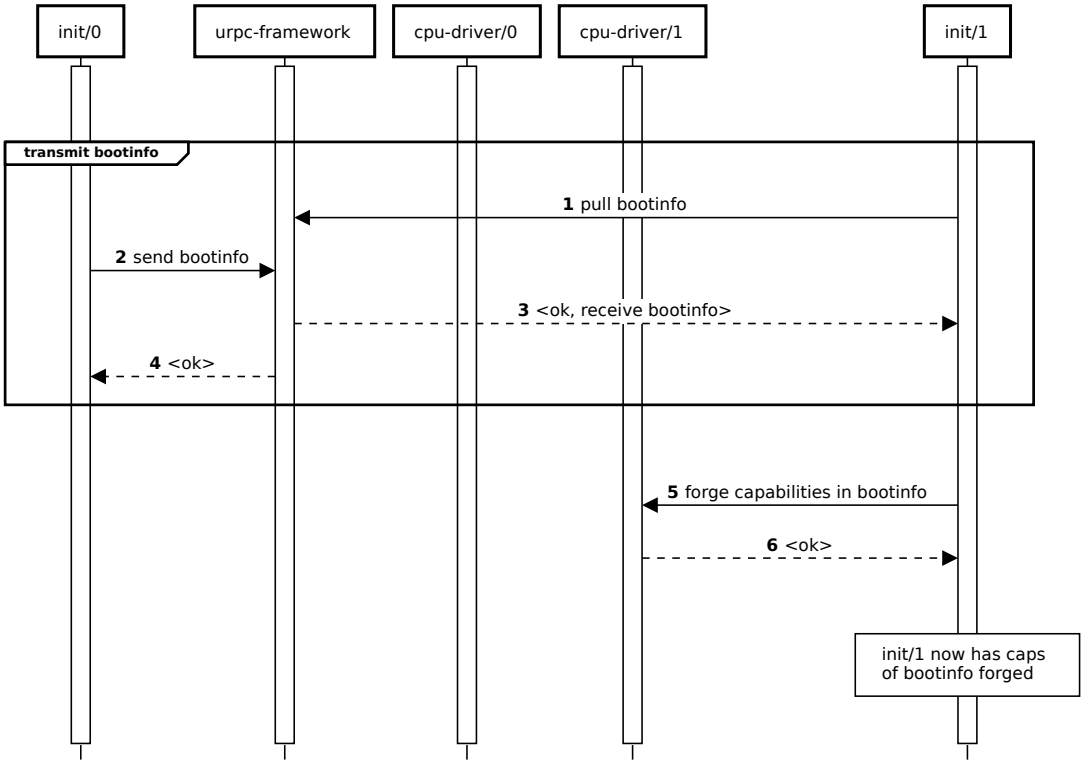

Figure 5.3.: Transmission of bootinfo from core 0 to core 1

This naive nature of the design is on purpose and is planned to be replaced by a ring buffer and a more generic URPC framework in the next milestone. It also acts as a baseline for performance comparisons we conduct.

5.2.2. Barriers

Due to ARM's weak memory model we cannot count on stores being observed by the other core in the same order that we write them down. With memory fences we enforce ordering constraint on memory operations. Barriers are further explained in the next chapter in section 6.2.1.

```
struct urpc_shared_mem {
    enum urpc_status status;  /* 0: empty, 1: master, 2: slave */
    enum urpc_msg_type type;  /* 0: BootInfo, 1: SpawnRequest, ... */

    union {
        /* data structure to transmit bootinfo */
        /* data structure to transmit spawn request */
        /* ... */
    };
};
```

Listing 6: Data structure for message exchange on URPC Frame. The first message to be transmitted is information about bootinfo. A status indicates for which party the chunk is and a type describes the memory layout of the chunk.

5.3. Memory Allocations on Core 1

With the `bootinfo struct` at hand on `init/1` we can now forge capabilities to RAM on our hardware. There are several designs how to arrange memory between cores. Barrier are further explained in the refined framework.

Request Memory on Demand In this design we implement on demand memory via the URPC channel. `init/1` can request memory as needed from `init/0`. The URPC channel transmit a `base` address and `size` and forges new capabilities on demand. There is one memory manager on `init/0`. This design easily scales to multiple cores. It provides optimal memory allocation as we only provide what is requested. However, it does not allow `init/0` or `core 0` respectively to go offline. It requires more engineering effort to implement as we have to provide allocation functions which we expose via URPC.

Split Memory Regions in Half Before we add memory regions to the memory manager on `init/0`, we split all available memory in half and only add one half to the memory manager on `init/0`. The other half we transmit via URPC channel to the other core. In this design `init/0` and `init/1` run their own memory manager independent of each other. This design allows `core 0` and `core 1` to manage memory independently of each other (after bootstrap). Memory allocation is, however, not optimal as the system may run out of memory on `core 0` even if `core 1` still has plenty of free memory available.

Due to time constraints, we decided to implement the design with the least engineering efforts. We opted for *Split Memory Regions in Half*. This comes with the implication that we can not dynamically spawn additional cores beside `core 1` and memory may be inefficiently distributed. Switching to the design *Request Memory on Demand* can be implemented later on with reasonable efforts if the inefficient distribution of memory causes problems. It is further not a requirement to support more than two cores.

5.4. Spawn Dispatchers on Core 1

In order to spawn dispatchers on `core 1` we use the `urpc-framework` functions to instruct the process server running on `core 1` to spawn. `Core 1` runs a slave version of the process server which does not keep track of spawned PIDs and simple follows orders to spawn dispatchers. Figure 5.4 shows steps to spawn `dummy/1`.

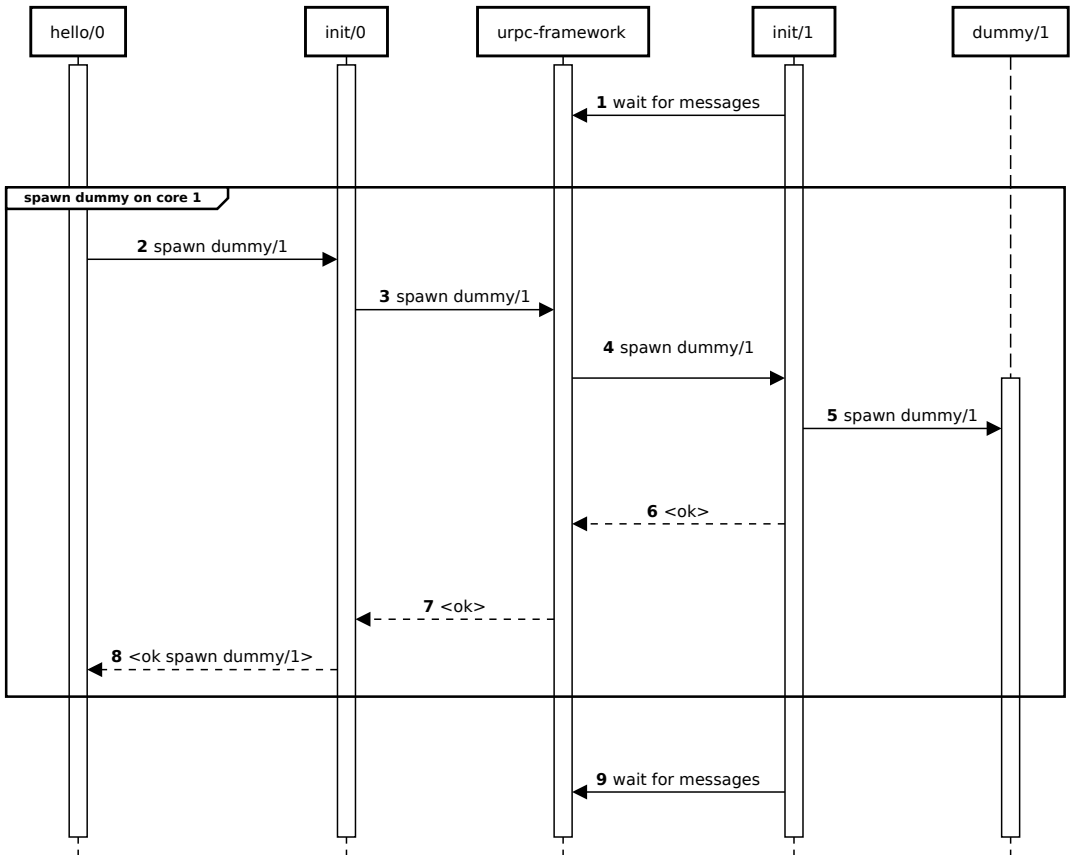

Figure 5.4.: Master/Slave message exchange to spawn a dispatcher

(a) `hello/0` uses an LMP channel to talk to the process server running on `init/0` to spawn `dummy/0`.

(b) The process server on `init/0` decides based on the given `core id` to use the urpc channel to spawn `dummy` on `core 1`. Process server on `init/0` is in master role. Upon receive of a response from slave, `init/0` is blocked and cannot receive other LMP messages. This greatly simplifies the design as we do not have to store and queue requests but is slow.

(c) The process server in `init/1` listens for messages in a thread and receives a spawn request to spawn `dummy/1`.

(d) Status is reported back via the urpc channel to `init/0` and back to `hello/0` via LMP channel. `init/0` unblocks and can receive new requests.

5.4.1. Debugging Paging System

Introduction of an additional thread in `init/1` puts our code base to the test. We quickly realized that the LMP framework implemented in Chapter 3 is not yet thread safe. Thread safety was not considered when implementing the `aos_rpc_*` APIs. We belive threads are a useful layer of abstraction and should be supported by an operating system. We split our engineering efforts and one team introduced proper locking of LMP code. The paging system caused more troubles with thread safety. Upon introduction of multiple threads in `init/1` we noticed deadlocks in paging code. Bugs were based on race conditions not obvious to debug and were not reproducible. We debugged for several days and due to time constraints decided to drop multi-threading support in the URPC framework. This once again reminded us that parallel programming is hard. Dropping threads prevented race conditions, however, did not fix bugs. In the next milestones we are going to spend a lot more time fixing paging issues.

5.4.2. Single Threaded URPC Framework

The changed design includes non blocking lookup if messages are available. We moved polling within the thread into the main thread of the dispatcher (the thread which loops on the default waitset). `lib/aos/waitset.c` supports a non blocking event dispatch. There are likely more efficient ways to implement this as we now busy loop through all of the time chunk until preempt. Given the naive nature of the implementation this decision is justified.

```
struct waitset *default_ws = get_default_waitset();
while (true) {
    /* serve urpc */
    err = urpc_slave_serve_non_block();
    if (err != LIB_ERR_NO_EVENT && err_is_fail(err)) { /* err */ }

    /* serve lmp and everything else */
    err = event_dispatch_non_block(default_ws);
    if (err != LIB_ERR_NO_EVENT &&  err_is_fail(err)) { /* err */ }
}
```

Listing 7: Single threaded main loop in init/1. It is checked in a non blocking manner if new messages are available.

5.5. Thread-Level Synchronization

Since the description of this milestone contained a recommendation for getting threading to work, we wanted to start making our modules thread-safe. The modules we identified as needing to be synchronized were

- `morecore`
- `paging`
- and `aos_rpc`.

`morecore` and `paging` need to be synchronized since there exists only one instance of each per domain, while `aos_rpc` needs to be synchronized firstly during channel acquisition and secondly per channel, since each channel (init, process, serial, memory) exists only once per domain and is thus also shared between threads.

5.5.1. Dependencies between Modules

All three modules interact with each other, which makes synchronizing them a bit tricky. `morecore` depends on paging to feed new memory to `malloc()`, `paging` uses `malloc()` for all its data structures and does slab refills for the range trackers, which means it depends on `morecore` `aos_rpc` since slab refills work through the memory server. `aos_rpc` depends on `morecore` and `paging`, since it uses `malloc()` to allocate variable-sized RPC messages and `slot_alloc()` which uses paging calls.
After adding mutexes to each of these modules to protect their respective data structures, we immediately encountered our first deadlocks. Soon, we discovered that the calls between these modules were often done in critical sections, which means that the modules depend on each other while holding their respective mutexes. To give a better overview, the dependencies between the modules are shown in figure 5.5.

5.5.2. Eliminating potential Deadlocks

We realized we had to move dependencies out of the critical sections to eliminate the risk of deadlocks. We notice that we can easily move the calls in `aos_rpc` out of its critical section, at least for requests going to the memory server where the message sizes are fixed. For that we introduce a way of sending a message without using `malloc()` and where the receive capability slot is allocated before the critical section. In `aos_rpc` we can simply move the slab refill out of the critical section. However, moving the `malloc()` calls out is not as trivial, since it is not apparent how much memory will be needed for a given critical section and furthermore, the hash table implementation uses `malloc()` internally.
One possible solution we identified to solve this issue was to not use `malloc()` in `paging`. Arguably, we should not have relied on it here in the first place. This would have required us to use slab allocators throughout the paging module or implement our own `malloc()`-equivalent for internal use in the paging module. The hash table implementation would

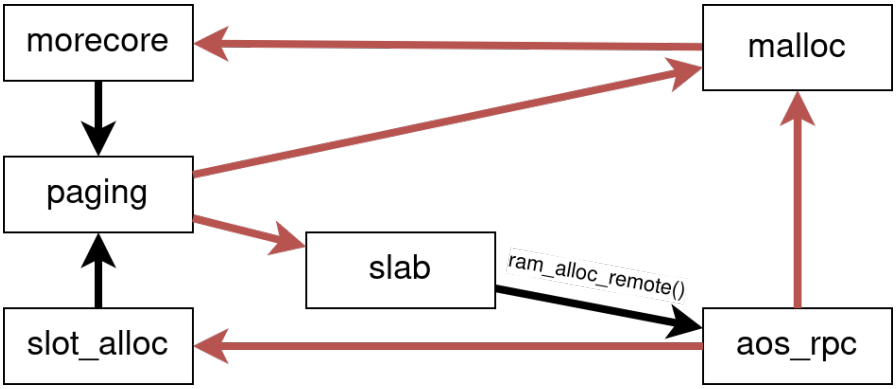

Figure 5.5.: The relationship of the modules that need to be synchronized. The arrows show the call-dependencies between the modules. A red arrow indicates that the call happens in a critical section (i.e. while holding the mutex of the respective module).

have also needed to be customized. Another solution would have been to unlock the mutex just for the calls to other modules. This is what the `slot_alloc` module does to call paging code, but in our case this could have led to inconsistent paging states and thus was not feasible.
Instead, we opted for an easier solution after the time we already invested into understanding the whole situation, which is to share a mutex between `paging`, `morecore`, and `malloc()`. We are aware that this is not the most elegant solution, but given our time constraints it seemed like the best way for us.
Generally, we can conclude that any two modules which each use their own mutex for synchronization are a potential for a deadlock as soon as they call each other from their critical sections. This might seem obvious since this is essentially the definition of a deadlock, but having complex module dependencies like in our case makes it tricky to anticipate some of these deadlock situations. As an additional challenge, when we began working on this we were not consciously aware of all dependencies between the relevant modules.
To test our thread-safety we wrote a multi-threading test that put our paging code to its limit. Despite our efforts, it still failed under very harsh conditions, but we left it at that for now and hoped that it would not become a problem in practice. If it did, we will have to fix that later.

5.6. Optimizations and Improvements

Multiple Slots We currently serialize all requests and use a single chunk of memory in the urpc frame. This is not efficient but is justified for the naive nature of the urpc framework. We are going to change this in the refined framework in the next chapter.

Thread Support Recall bugs in multi-threading support in section 5.4.2. While a usable operating system can be implemented with events and waitsets only, threads provide a useful layer of abstraction which is why we should fix the current threading bugs. The next chapter in section 6.5 will talk about this in more detail.

URPC channel type Instead of using threads we can change explicit peeking of URPC events into channels and waitsets. We opted for explicit peeking in the main loop because it was more straightforward to implement and we were not familiar with functions in `waitset.c`. This enhancement includes adding a hook in `waitset.c` such that we can poll for URPC events when polling channels for events in `waitset.c#poll_channels_disabled(handle)`. If we do not implement this we should at least call `thread_yield()` such that we don't busy loop.

Generalized URPC The current implementation can not transmit arbitrary content via URPC. Introduce a more generic URPC framework which accepts a `size` and a `pointer` and does the marshalling on both ends. This is going to be tackled in the next milestone.

6. URPC Message Passing

Did you test the code? – Yes... well... I compiled it.
— Conversation during this course

With support for multiple cores in place we can now refine the user-level messaging protocol introduced in the last chapter.

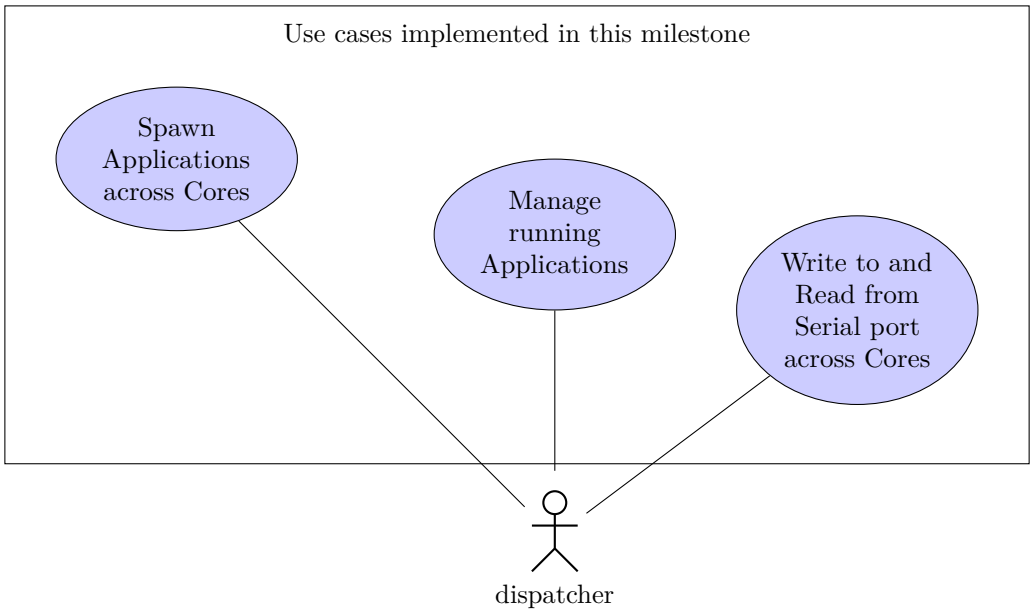

Figure 6.1.: Use cases covered in this chapter. An arbitrary dispatcher can access functionalities across cores

Recall that we already implemented most of the functionality in previous milestones with the exception that it was only accessible on a single core (namely core 0 within the init dispatcher). In the next sections we will describe our design decisions which introduce a monitor to coordinate and delegate tasks across cores. It should be noted that our monitor is only loosely related to the monitor dispatcher used in mainline Barrelfish. In section 6.2 we will write about the refined URPC library. We will conduct performance evaluations on the new design architecture. The new library along with the monitor introduce performance implications which we shall see affect responsiveness of the terminal server rather substantially. Furthermore, the monitor makes use of threads which put the thread safety of the system once again to the test.

6.1. Introduction of a Monitor

When opting for an initial design we try to keep the following principles in mind.

Uniformity Favor uniformity more than speed. Do not introduce exceptions in the architecture which break abstraction. While the system should not be pathologically slow, the initial version should work and be a baseline for further optimizations. Changes to increase speed can be done as enhancements later on.

Extensibility The individual milestones will likely provide RPC services across cores. Make it easy to extend the monitor with more functionality.

KISS Despite extensibility keep the initial version simple and stupid. Do not over engineer. We have pending bugs in the paging system which have high priority.

After much delibaration, these design principles lead to a monitor component which runs within `init/0` and `init/1`. Recall that `init/0` already runs all `aos_rpc_*` services from previous milestones. The following are the important aspects of our architecture.

- In order to access `aos_rpc_*` functionality across cores, we deploy a monitor on both cores. figure 6.2 gives an high-level overview of the resulting architecture. A more detailed view of the architecture is presented in figure 6.3.
- A monitor acts as a single point of service for dispatchers to access `aos_rpc_*` functionality. It is accessed via LMP interfaces.
- The monitor knows on which core a `aos_rpc_*` service runs. It forwards the request to the dispatcher which runs the corresponding `aos_rpc_*` service. The monitor has URPC connections to all such services.
- In our setup the serial, init, process and memory server run within `init/0`. There is also a memory server deployed within `init/1` (see reasoning behind this in section 6.3.2).
- The monitor uses an URPC connection even if the targeting service runs within the same core, or in our case within the same dispatcher. This follows the Uniformity principle described in section 6.1.
- A generalized URPC library allows marshalling and transmission of arbitrary messages. If capabilities are transmitted they are automatically forged on the other core. See section 6.2 for details.
- The monitor handles local tasks for the core it is running on. The concept of local tasks will be further introduced in section 6.1.2.

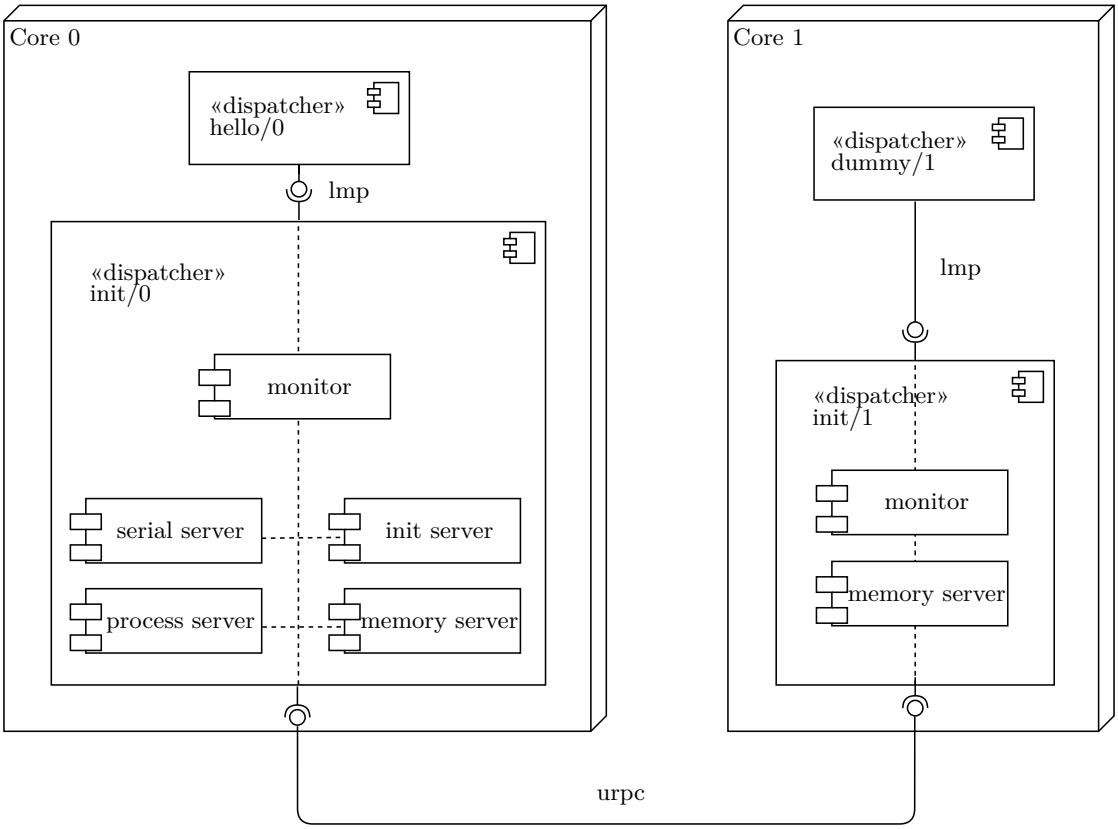

Figure 6.2.: Architecture Diagram showing the high-level architecture of two sample applications hello/0 and dummy/1 which access common services across two init dispatchers

6.1.1. Detailed Architecture

We deploy `aos_rpc_*` services within `init/0`. This is motivated by KISS (section 6.1). Our architecture allows deployment of these services within separate dispatchers but we currently see no reason to. The monitor exposes an LMP interface for clients to call. In figure 6.3 this is marked with a, and b. The LMP side of the monitor was straight-forward to implement. We could conveniently reuse all of our LMP code. Client dispatchers receive the capability to the LMP endpoint during spawn.
The monitor forwards requests to the correct `aos_rpc_*` service depending on the field `enum rpc_message_method` in the payload. This is marked with e for `init/0` and f for `init/1`. The `aos_rpc_*` services process the request and reply back to the monitor (e, f) which then unblocks and replies the response back to the caller (a, b). The monitor receives events in a separate waitset and runs in a separate thread. It does not block the rest of the dispatcher.

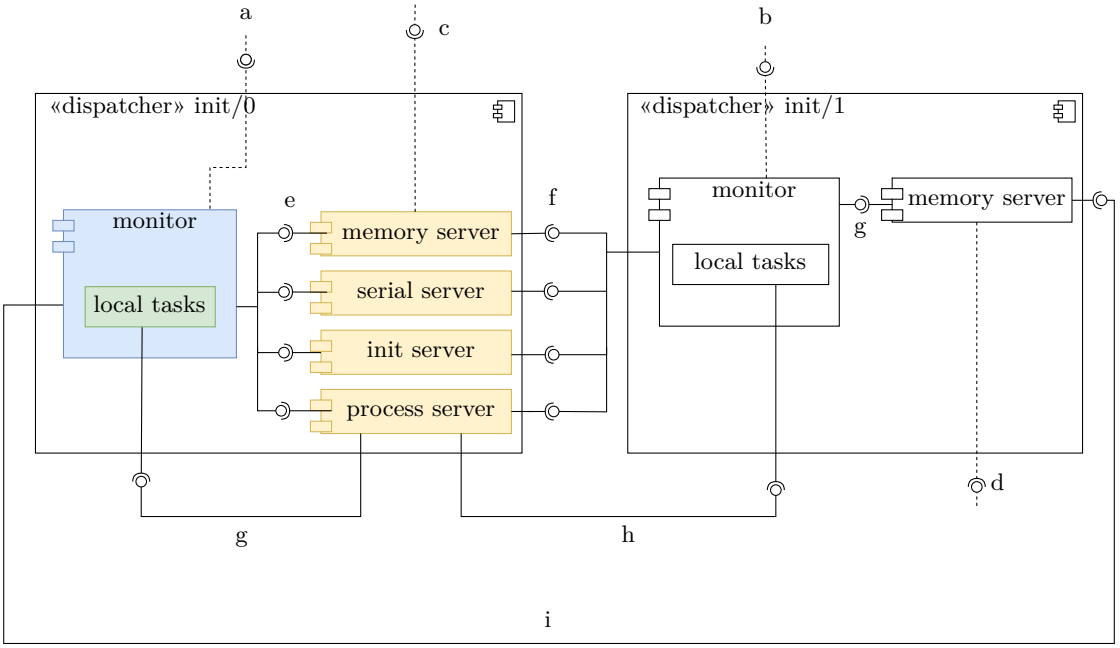

Figure 6.3.: URPC architecture with monitors. Monitors runs within init/0 and init/1. Exposed LMP interfaces (dotted lines): a, b, c, d. Exposed URPC interfaces (straight lines): e, f, g, h, i. Dispatchers on core 0 and 1 either call LMP interfaces a, c or b, d respectively. Color coded are threads within init/0. Yellow: thread 0 on default waitset, blue: thread 1 on monitor waitset, green: thread 2 on local tasks waitset.

6.1.2. Local Tasks

Some task must be executed on a core which does not run an `aos_rpc_*` service. This currently includes spawning a dispatcher. `init/0` runs the process server and can spawn on core 1. This is where *local tasks* come into play. The process server sends a local task to the monitor of the other core which executes the task—in this case spawning a dispatcher—on behalf of the process server. The monitor then replies with a status. This is again motivated by the KISS principle.

6.1.3. Limitations

Monitor is Serializer The monitor acts as a single point of access. It is involved in each request and a consecutive requests can only be served if the previous request has returned. With only a few dispatchers running on a core and under low load this works fine but it does not scale well with increasing number of dispatchers. Even if `aos_rpc_*` services are deployed on separate dispatchers, every request is pushed through the monitor.

Within workload of this course scaling is not an issue. A better design nonetheless removes the monitor out of the loop once connection to an `aos_rpc_*` service is established. This can be achieved by sharing an URPC frame between service and client dispatcher directly. The monitor then acts as a gateway to setup connections. We considered this design initially but we opted for KISS and implemented what seemed most straight forward.

Recompilation for new Services The monitor does demultiplexing of requests based on their method. There is currently no Nameserver functionality involved. To support new services or *Local Tasks* the monitor must be recompiled. Without Nameserver functionality this is unavoidable.

6.2. Refined URPC library

Our goal with this library was to provide similar functionality as our LMP channel wrapper, which implements the segmentation for transmitting arbitrarily long RPC messages. The difference is that in this case the underlying communication takes place over a shared memory region. It is the lowest-level interface to what we call an UMP channel. A UMP channel uses a single frame for bi-directional communication. The frame has to be of a fixed size predefined by the URPC library. The frame is split in half, one half being used for one direction of communication. The library does not offer functionality to transfer this shared frame between parties. This has to be done using any other kind of communication.
We had two problems to solve for our design.

1. Specify for each side which half of the frame to use for transmission and which half to use for receiving from the other side, since that assignment has to be reversed between both sides.

2. Specify where the underlying frame is being initialized (zeroed).

To solve both of these problems, we decided to define two roles: the *establisher* and the *non-establisher*. The establisher allocates the shared frame, initializes it and uses the upper half of the frame for transmitting and the lower half of the frame for receiving data. The non-establisher assumes that the frame is already initialized and uses the lower half of the frame for transmitting and the upper half of the frame for receiving data. Both sides have to initialize a channel passing the shared frame and their role to the URPC library. When programming using this interface it is always clear what role to use, meaning it is not information that itself has to be transmitted over the UMP channel. Since the establisher initializes (zeros) the frame it is important that the frame is not being initialized on the other side before that happens. To ensure that, the shared frame should only be transferred after is has been initialized by the establisher. Otherwise there could be left-over data in the shared frame that the other side interprets as new data to read. One could have the idea to simply let the non-establisher initialize the

shared frame, but that would result in a similar problem. In that case, the establisher could have already written data into the frame which it assumes to be read by the other side. If the non-establisher would initialize the frame, however, such data would be lost. Each half of the frame contains a queue of cache-line-sized slots and establisher and non-establisher both store pointer to the next slot to read from or write to, as described in the assignment. Each slot uses one word to indicate whether a slot is free or used, two words for capability transfers (more about that section 6.2.2), and four of the remaining words for data transmission to keep it similar to the LMP communication. The whole scenario is illustrated in figure 6.4.

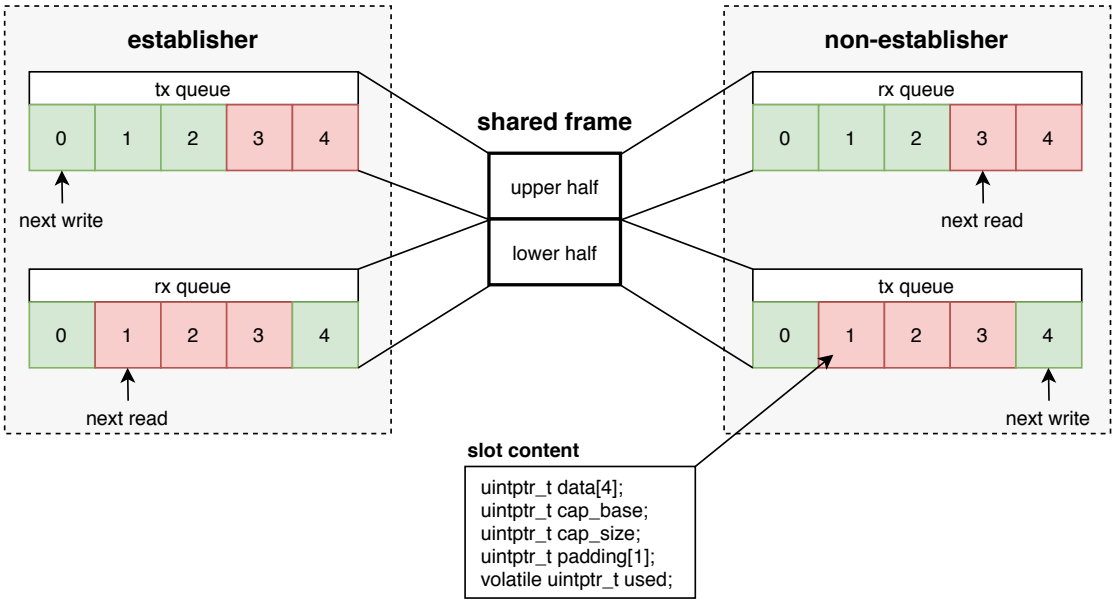

Figure 6.4.: The usage of the shared frame for UMP channels. Green and red indicate free and used slots (cache-lines) respectively.

6.2.1. Synchronization

Because we do not synchronize the UMP channels between dispatchers, a UMP channel can only be used for 1-to-1 communication. This is in contrast to LMP channels, where open-receive channels can be used for n-to-1 communication. To send an RPC message, the message is segmented into pieces small enough for a the cache-lines and the library blocks until the full message has been transmitted. To receive an RPC message we implemented a non-blocking and blocking call. The non-blocking call returns immediately if there is no data ready to receive, while the blocking call will block until it received a full RPC message. However, if an RPC message is started to be received, the call always blocks until the full RPC message has been received. To ensure thread-safety the channels are synchronized with a mutex similar to our LMP wrapper. As a result, it is not possible for two messages to become interlaced on the shared frame. Furthermore,

the library itself does not provide a mechanism to notify a caller asynchronously as soon as a message is received. This behavior can be implemented by creating a thread that repeatedly calls one of the receive functions.

Data Barriers

Because of ARM's weak memory model we need to utilize data barriers. In total we need to use four barriers. During sending we firstly need a barrier to ensure that the data is not written before the used flag of the next slot is set to free since otherwise the new data could be written before the other side has fully read the data that was still occupying the slot. Secondly, we need a barrier to ensure the data is written into a slot before its used flag is set since otherwise the receiver may read the data before is has been fully written. Similarly, during receiving we need a barrier between checking the used flag and reading the data and another one between reading the data and setting the used flag to free. For all of these we used a full system data memory barrier (`dmb sy`) as suggested in the assignment. The exact procedure for writing to and reading from a slot is shown in listing 8.

```
// Wait until the next tx slot is free
while(slot->used == 1) {
    thread_yield();
}

BARRIER_DATA;

// Write to slot
...

BARRIER_DATA;

// Set slot to used
slot->used = 1;
```

```
// Block until the next rx slot is used
while (slot->used == 0) {
    thread_yield();
}

BARRIER_DATA;

// Read from slot
...

BARRIER_DATA;

// Set slot to free
slot->used = 0;
```

Listing 8: Code structure for writing to (left) and reading from (right) a slot in a shared frame. `BARRIER_DATA` is a macro for the `dmb sy` assembly instruction.

6.2.2. Transferring Capabilities

We cannot transfer capabilities directly transferred over a UMP channel. Instead, we limit our UMP channels to transmitting only frame capabilities and use two dedicated words in the slots to transfer their "meta-data", namely base address and size. When a receiving side notices that those words are not zero, it will forge a corresponding frame capability and pass it on to the caller of the receive. This works similar to LMP channels, where each LMP message may carry a capability with it, with the difference that our UMP channels can only transfer frame capabilities.

6.3. Accessing System Services across Cores

With the URPC library, the monitor and local tasks in place we can now implement communication with our existing system services across cores. Specifically, it should be possible to centralize the system services of which we had a separate instance for each core up to this point. To do that we migrate the LMP servers to the new UMP channels since LMP cannot be used to communicate between cores. We wrote a generic UMP server, which has essentially the same API as our generic LMP server. This made the migration pretty quick and painless.
There are two notable differences compared to the LMP server.

1. The UMP servers are not integrated into the waitset subsystem. Instead, each server has a serve-function, which each has to be called alongside the `event_dispatch()` function in the main message loop. Firstly, this means that we now have to use the non-blocking variant of `event_dispatch()`. Secondly, this means we have to be careful while serving a request of one server to not send requests to another server, since that would lead to a deadlock. This would not have been a problem if we used a separate thread to serve the requests for each server, but we were not confident enough in the thread-safety of all of our modules for that.

2. As explained in section 6.2.1 a UMP channel is restricted to 1-to-1 communication. This means UMP servers cannot provide an open-receive channel that new clients can use to open a connection. Instead, to add a client to a UMP servers a function has to be called locally that adds a client to the client list of the UMP server. The function takes a channel as an argument which the new client can then use as a communication channel with the server.

Because of the second difference we statically setup all UMP channels and pass them to the second core during initialization. That includes a UMP channel to each server and a UMP channel that the monitor on the second core will use to receive local tasks.
As previously explained, an RPC call by a dispatcher to one of our servers now works by sending an LMP request to the monitor of the core the dispatcher is running on. The general sequence of messages is shown in figure 6.5.

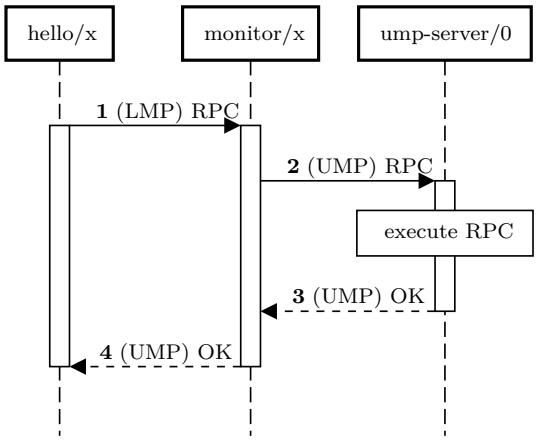

Figure 6.5.: General message sequence for a process running on core x to send an RPC to a system service.

We then started to eliminate the redundant instances of our system services. The init server and the serial server did not require any special changes other than migrating them to the UMP servers and launching an instance only on core 0. The process server and the memory server, on the other hand, required more attention, which is explained in the following sections.

6.3.1. Process Server

The process server had to be extended to send local tasks to a monitor. Which monitor depends on which core it wants to spawn a new dispatcher. Figure 6.6 shows the sequence of messages if a process on one core wants to spawn a process on another core. It can be compared to the sequence of the naive URPC library shown in figure 5.4. The process server receives a response after the local task has been executed. On success, the list of spawned dispatchers is updated. In any case, a response is sent back to monitor that forwarded the spawn request to the process server, which then replies back to the original requesting dispatcher through LMP.

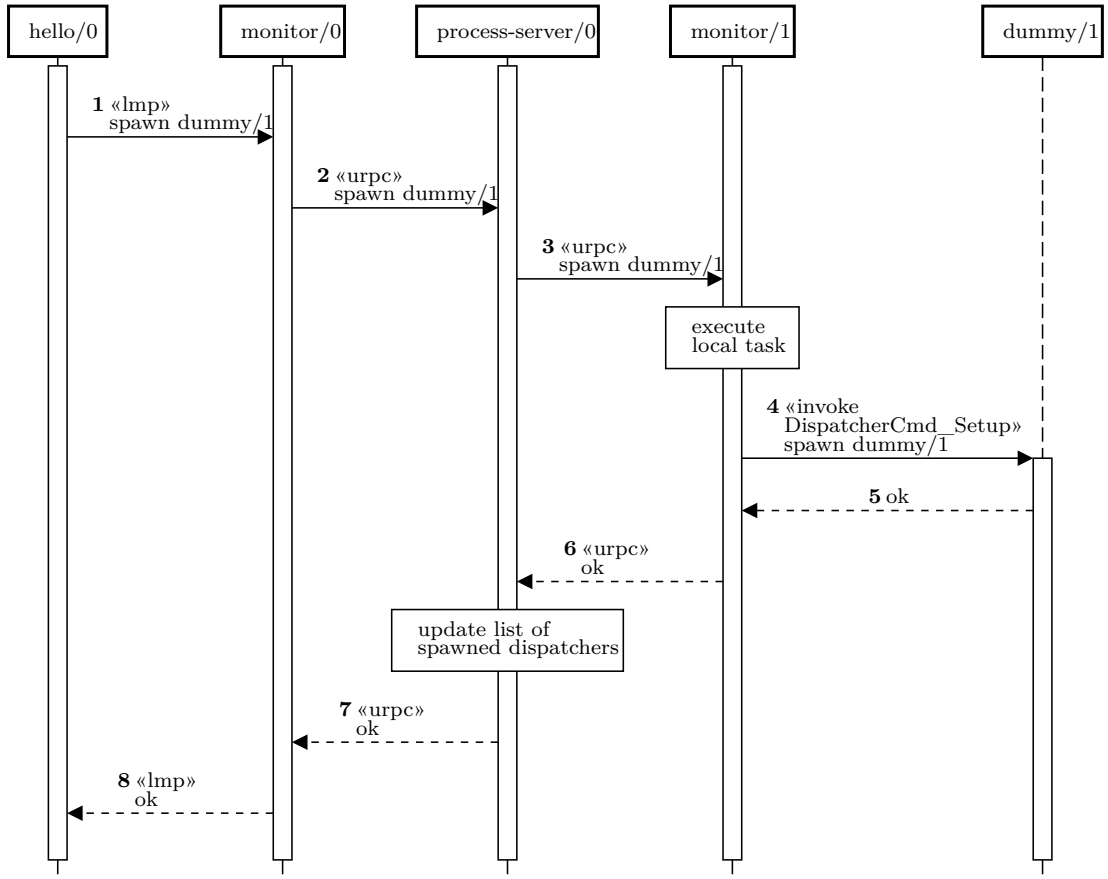

Figure 6.6.: Message sequence for hello/0 invokes local task in monitor/1 to spawn dummy/1

6.3.2. Memory Server

Unfortunately, centralizing the memory server comes with a few hurdles because of the design of our monitor server. Since the monitor server blocks until it receives a response for a forwarded request, there are many deadlock situations created if all requests to the memory server have to go through the monitor which blocks until it receives a response. For that reason, we decided to treat the memory server as a special case and stick with the current design, where each core has its own memory server running in the init dispatcher and processes use LMP to allocate more memory.

6.3.3. Performance Drawbacks

Because the monitors are acting as the only way for dispatchers to communicate across cores, many processes using it at the same time obviously quickly creates a performance problem. The fact that the monitor is blocked until it receives a response for every single request that goes through it certainly does not help either. This reinforces our decision

in the previous section for treating the memory server as a special case. Even if it was easily possible to migrate the memory server to the new architecture, performance would also be atrocious, since it is so frequently used.

6.4. Performance Measurements

We conducted a few measurements to compare our different implementations of RPCs. The results can be seen in figure 6.7.

6.4.1. Measurement Setup

We measured execution time of the following RPCs:

- `aos_rpc_send_number()`, which sends a number to init/0 and does not return with a response. Execution time starts with invoke of the `aos_rpc` wrappers and ends in the grading callbacks on the servers.
- `aos_rpc_process_spawn()`, which spawns a domain and returns with a response. Execution time measures round trip time of the `aos_rpc` wrappers.

The binary to spawn performs a `debug_printf` and simply returns. These two RPCs are benchmarked throughout the course. Firstly, in their initial implementation based on LMP (chapter 3), next in a naive version across cores (section 5.4.2), thirdly, in a refined version across cores (chapter 6), and finally, in the version which is submitted with the report.

6.4.2. Discussion

As expected, LMP is the fastest implementation and our initial URPC implementation is faster than our more sophisticated URPC implementation. Once the Nameserver had been implemented in a later chapter, direct UMP channels could be established which improved performance considerably. For spawning the LMP performance shows that a large chunk of the execution time in that case is spent waiting for the actual spawning to finish, not the message transmission. Note that the difference between sending a number over LMP and direct UMP is still very large even though the difference is not well shown by the graph (see caption for exact numbers).

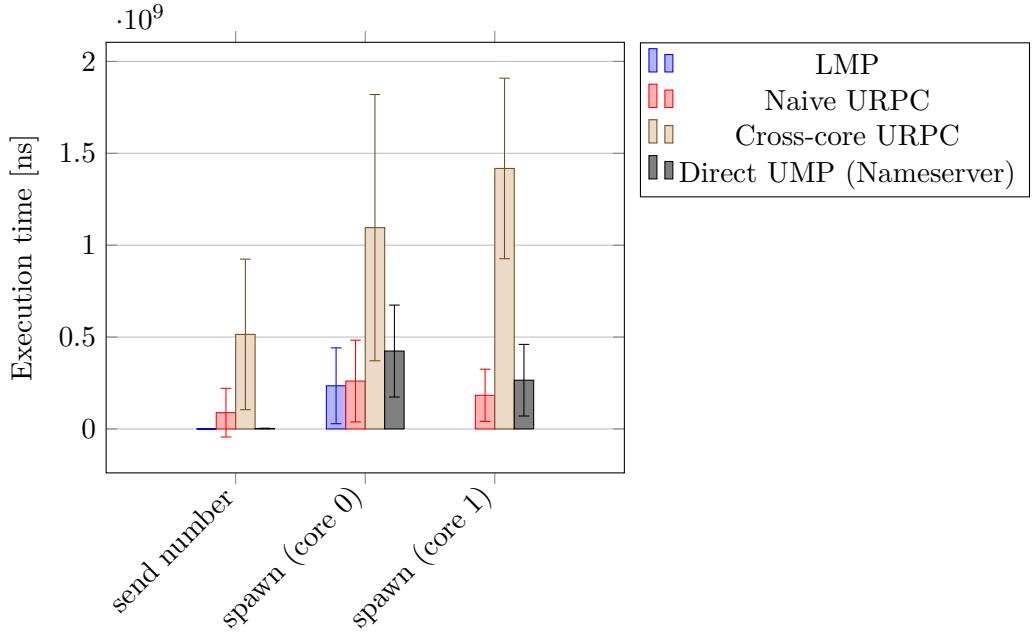

Figure 6.7.: Performance evaluation of our different RPC implementations. The RPCs that where tested are: `aos_rpc_send_number()`, `aos_rpc_process_spawn()` with core 0 as target, and `aos_rpc_process_spawn()` with core 1 as target. Error bars represent one standard deviation. The mean when sending a number using LMP was 15 100 ns (σ =3272 ns) and using direct UMP was 2 635 011 ns (σ =1 103 092 ns). Note that LMP cannot spawn on core 1 and that this graph also includes the implementation that makes use of the Nameserver which has been added later.

6.5. Debugging Paging System

To all of our disappointment, the known issue in the thread-safety of the paging module (see section 5.5.2) came here to light in practice, because of the threads used in the monitor. We tried once again to track down the last problems there. As an initial success, we found a rather minimal configuration of our multi-threading test to reproduce the issue. At some point we managed to narrow it down to the implementation of `free()`, which interacts in a quirky way with our `morecore` implementation, but we were unable to identify what exactly was going wrong there. Our best guess was that the problem emerged from our static/dynamic heap construct in `morecore` and the interaction between them in `free()` when multiple threads are running. We saw a high probability that eliminating the distinction of static and dynamic heap would make the problems disappear. To do that, however, we would have to eliminate the uses of `malloc()` in the paging module and replace them with something else. Unfortunately, since the deadline was drawing nearer and without any new progress in our endeavours we were forced to find another work-around for our dire situation.

6.5.1. Drop Ability to Free Memory

We noticed that the issues disappeared if we changed `free()` to immediately return (i.e. do nothing). This was a quick work-around and since our system did not allocate that much memory anyway it did not have a noticeable impact on our—admittedly limited—use-cases. Another reason for why this is not as big of a deal as it might appear is that `free()` did not do that much in the first place. Since we have no way of giving back memory to the memory server, the most `free()` can do is to give back memory for reuse in the same dispatcher that it was allocated in. Still, we acknowledge that this was a work-around, not a solution for our threading issues and we did not want to leave it like that. Moving forward, we anticipated that we will either have to find a solution to the threading issues or not depend on threads for the remaining tasks of the project.
Note that removing the thread from the monitor is not so trivial since it would create a deadlock situation as described in section 6.3 if a dispatcher wants to spawn another dispatcher on the core on which the process server is running. In that case the process server, while processing a spawn request, would send a local task request to the monitor server, which is being served in the same loop.

6.6. Optimizations and Improvements

A Performance problems as explained in section 6.3.3.

B Free should be re-activated (see section 6.5.1).

7. Common Changes in Architecture

To start, press any key. Where's the "any" key?!

— Homer Simpson

Unfortunately, we were unable to fix our remaining threading issues, which means we decided to not depend on threads for the remainder of the project. In the places where threads were in use we replaced them using periodic events in the default waitset. This had implication on design. Long running operations could not be blocking and had to store state and periodically check and resume if available.

8. Nameserver

Looking at code you wrote more than two weeks ago is like looking at code you are seeing for the first time.

— Dan Hurvitz

Individual Milestone: Christian Leopoldseder

The Nameserver is supposed to take care of setting up communication channels between dispatchers. Generally, with the Nameserver it should be possible for dispatchers to offer certain functionality which will be advertised system-wide to other dispatchers. Other dispatchers should then be able to open a communication channel to make use of the offered functionality. The uses-cases relevant to the Nameserver are shown in figure 8.1. The architecture into which those requirements were translated into is explained in the following section.

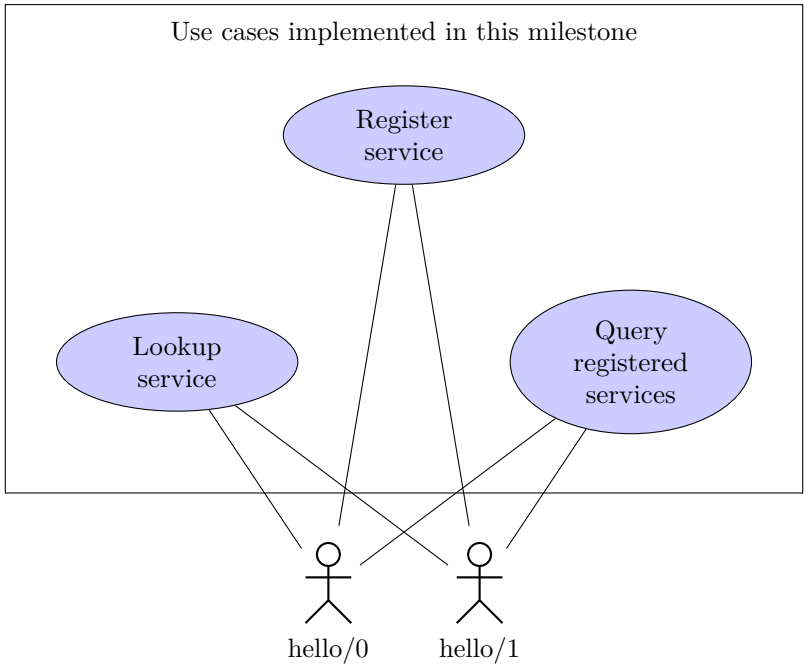

Figure 8.1.: Use cases relevant to the Nameserver. Actors are dispatchers running on either core.

8.1. Architecture and Implementation

The Nameserver is implemented as a UMP server and dispatchers communicate with it only indirectly over their respective monitor in the same way they communicated with all system services up to this point (see chapter 6). With the Nameserver, however, it is possible for all dispatchers to communicate with any server that is registered at the Nameserver directly over a UMP channel. This is in contrast to our solution without Nameserver, where dispatchers route all their requests over the monitor via LMP, which forwarded it the the respective recipient over UMP (see chapter 6). The following sections will go through the use-cases and explain how the architecture handles each one.

8.1.1. Registration

With the Nameserver, we define the term "service" as a dispatcher that registers itself at the Nameserver and handles incoming service requests. When a dispatcher registers itself at the Nameserver it uses the `nameservice_register()` function. This call does not only send the register request to the Nameserver, but also initializes the local data structures necessary for a service. That includes initializing a UMP server for accepting service requests and a UMP channel to receive add-client requests. In addition, a periodic event is created to call the serve function of the UMP server and poll the UMP channel for add-client requests.
The service name, the underlying frame of the add-client UMP channel, and the PID of the dispatcher are transmitted to the Nameserver as part of the registration request. Upon receiving the registration request, the Nameserver initializes the add-client UMP channel using the received frame and adds it, together with the service name and the PID to its list of services.

8.1.2. Lookup

Before a dispatcher can use a service it has to look it up at the Nameserver. It can do that using the `nameservice_lookup()` call, passing a service name. When the Nameserver receives such a request, it looks up the service by name in its service list. It then uses the add-client channel to request a new UMP channel for that service. This add-client request is handled by the Nameserver library automatically by creating a new UMP channel and adding it to the client list of the UMP server. After the Nameserver received a reply that contains a new UMP channel (in the form of a frame capability) it forwards that together with the PID of the service to the dispatcher that sent the original lookup request. Note that because only the Nameserver has access to the add-client channel of a service, we are conforming to our 1-to-1 communication constraint of our UMP channel implementation (see section 6.2.1). As a result, every service can run in its own domain on any core and processes are able to acquire a direct UMP communication channel to any arbitrary service by using the lookup function. The procedure is illustrated in figure 8.2.

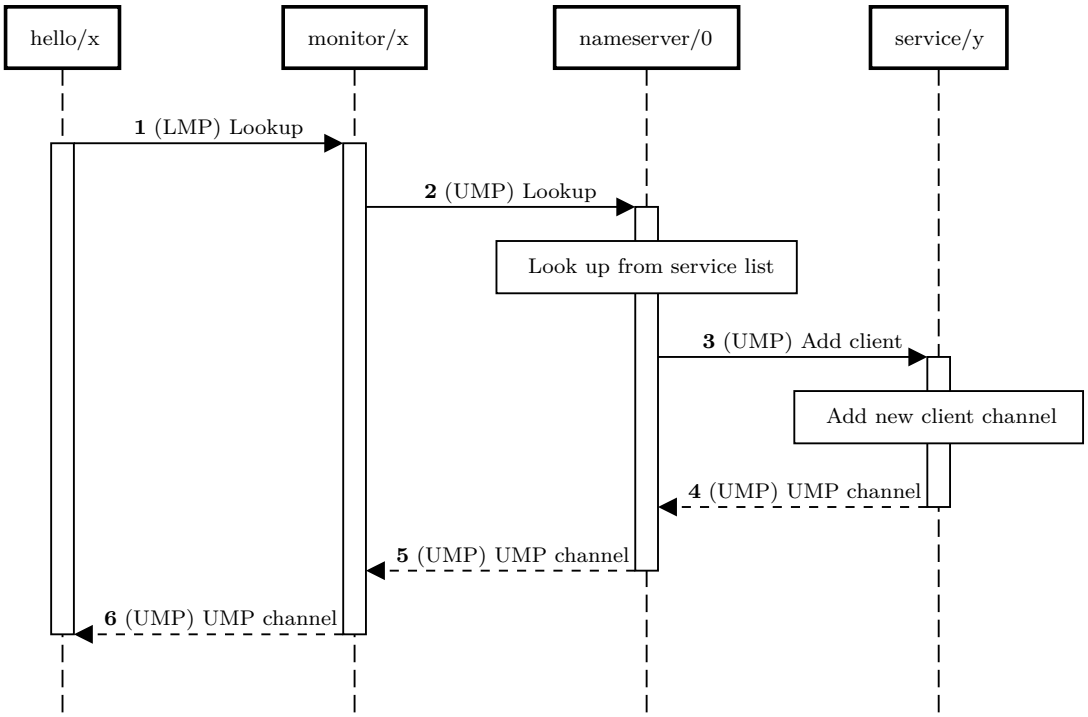

Figure 8.2.: Example of a dispatcher running on core x looking up a service offered by a dispatcher running on core y

8.1.3. Querying

To query the list of services registered at the Nameserver, the `nameservice_enumerate()` function can be used. A query is a string for which the Nameserver returns a list of all matching services in its service list. A service matches the query if its name starts with the query.

8.1.4. Sending Messages

To send a message to a service and receive a response, the `nameservice_rpc()` function can be used. Since that function uses raw buffers for the request and response data, the data is wrapped in RPC messages and then sent and received using our existing UMP library. It is also possible to send a message without expecting a response by passing `NULL` as the response buffer. The advantage of that is that the call will not block until a response has been received. The importance of that becomes more clearer in section 8.2.

8.1.5. Handling Service Messages

At registration, a service handler has to be passed that will generate a response based on a received message.

In our implementation the handler must allocate the response using `malloc()` since the Nameserver library has to take care of freeing the response buffer. The only alternative for implementing this without leaking memory would be to pass a fixed-sized buffer. That comes with the obvious disadvantages that responses would then be restricted to a fixed size.

8.2. Deadlock Situations

Since our threading issues remained unresolved (see section 6.5), everything had to be implemented without threads. Because of that, there are some situations that lead to a deadlock when using the Nameserver library incautiously. In general our system ends up in a deadlock situation if a process, makes a request (that expects a reply) to a service that ends up needed to be served by itself. The simplest case would be a dispatcher looking up or sending a request to a service running on the same dispatcher. But this also includes indirect loops, for example, a dispatcher sending a request to a service, which sends a request to another service which sends a request to the original dispatcher that started the RPC chain.
One implication of this is that the dispatcher in which the Nameserver is running (in our implementation the init dispatcher on core 0) must not use the standard functions of the Nameserver API to register or look up a service, since those will always end up at the Nameserver, which will not be served while waiting for a reply for either of those requests.
Furthermore, the init dispatcher also must not send requests to other services in general, since it has to be assumed that a service needs the memory server while handling a request, which cannot be served while the init dispatcher is waiting for a response message.
To avoid some deadlocks we also had to split up some serving functions that issue further RPCs during processing. For example, when the monitor server forwards a message it sends the the message and saves the context of the messsage. On each subsequent serving call it checks if a response is available and if so continues processing. This ensures that other tasks in the waitset are processed while the monitor server waits for a response. The same technique is used in the Nameserver when it sends an add-client request to a service and waits for its response.

8.3. Migrating the system services

Now that the Nameserver can be used to setup communication channels we could migrate our system services to the Nameserver API. They could (and should) even be put into their own processes and the forwarding logic in the monitor can be removed (except for the Nameserver). This considerably simplified setting up communication channels between our existing and soon to be added system components. Because of the direct communication channels, performance is also greatly improved as can be seen in figure 6.7.

8.3.1. Backwardscompatibility for RPC Calls

Even though the init, process, and serial servers were migrated to the Nameserver API, they should still be usable via the same `aos_rpc_*` interface. This was achieved, by using `nameservice_lookup()` in the `aos_rpc_get_*_channel()` functions and using `nameservice_rpc()` in the other `aos_rpc_*()` functions if a UMP channel is passed. Since `nameservice_rpc()` only takes raw buffers, not RPC messages, the RPC messages are put inside the buffer to be transmitted and read out of the buffer that will be received. That means, since internally the Nameserver API uses RPC messages for message passing, there are RPC messages transmitted were the payloads are their own RPC message. The advantage of this is that the existing system services could be migrated more easily, and the backwards-compatibility for `aos_rpc_*()` calls is ensured.

8.3.2. Process Server and Monitor

The process server still needed the monitors to execute local spawn tasks. The Nameserver simplifies that as well since the monitors can simply register themselves at the nameserver and then the process server can look up the respective monitor service for a given core to send it local spawn tasks.
There is a problem with that, however, because the monitor of the core where the Nameserver is running cannot register at the Nameserver in the usual way. If it did, it would create a deadlock situation as outlined in section 8.2 since they are both running in the same init dispatcher. Instead, we added some additional local registration functions to the Nameserver which the monitor running in the same dispatcher can use to register without creating a deadlock.
There were still tricky situations where deadlocks occurred that were related to this special local task interaction between the process server and the monitor server. For example, one of them occurred when one dispatcher does a lookup for the process server, while another dispatcher sends a spawn request to the process server at the same time. A deadlock occurred if those requests are processed in a particular inconvenient way:

1. The lookup request is received and forwarded by the monitor, which blocks the monitor
2. The spawn request is received by the process server, which sends a local spawn task to the monitor, which blocks the process server
3. The Nameserver receives the lookup request for the process server from the monitor and consequently sends a add-client request to the process server.

At this point, the nameserver will never get a response to the add-client request, because the process server is already waiting for another response itself. To give a better overview, the situation is shown in figure 8.3. To avoid this, we simply made sure that add-client requests are served while waiting for a response in any Nameserver API call.

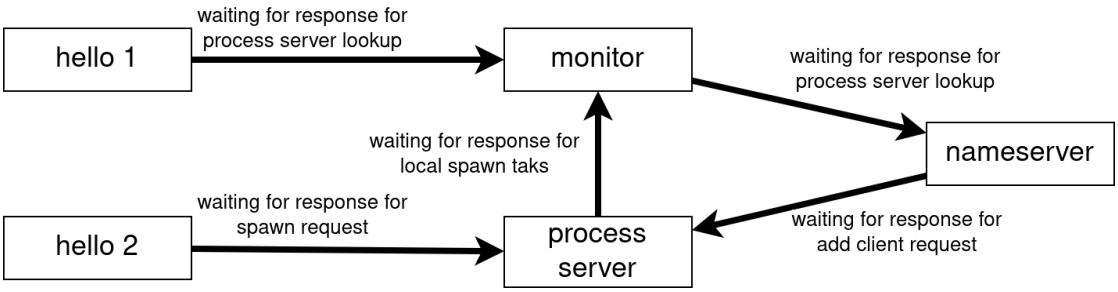

Figure 8.3.: Deadlock of the Nameserver subsystem. hello 2 sends a spawn request while hello 1 sends a lookup request, which could end up in this deadlock.

8.3.3. Memory Server

The memory server remains as the only service that is still only accessible core-locally and only over LMP. The reason for that is that migrating it to a newer interface like the other servers always came with a set of problems of which we thought the effort for solving them outweighed the eventual benefit. Still, we thought of some possible ways to achieve cross-core memory management, which are discussed in the following section.

Memory Management across Cores

Centralized approach. An obvious goal that one might like to achieve is centralized memory management, where all dispatchers obtain their memory from a central memory server. This kind of centralization could pose a challenge with our current architecture. For example, one could propose to run a memory server on only one core and register it at the Nameserver. Then all dispatchers do a look up for the memory service when they start and use that to acquire memory. That would be problematic as soon as another server, for example in our case the Nameserver, was running in the same dispatcher. Generally, processing a service request has to be assumed to allocate memory. In fact, just sending a response to a service request through the Nameserver API already requires an allocation on the heap as explained in section 8.1.5. This creates an implicit dependency on the dispatcher in which the memory server is running for every service that exists on the system. As discussed in section 8.2 issuing another RPC while processing a service request can be quite dangerous and as a consequence many new deadlock situations are created.
That problem could be solved by putting the memory server into its own dispatcher. That comes with its own problems, but they are easier to solve. For instance, a new RPC for adding memory regions to the memory server has to be implemented. Furthermore, there could be some problems with initializing whole system, since that could already require the memory server if the initial process RAM is not enough.

Distributed approach. Another simpler approach would be to implement means for distributed memory management. Each core could still have its own local memory server,

but they could still register at the Nameserver as a memory service. A core that is running low on memory could then do a `get_ram_cap()` RPC to get RAM from another core and add that as an additional region to the memory manager of the local memory server. This approach should be relatively easy to implement and should not create major problems since each core would still use the local memory server for the normal memory allocation.

8.4. Optimizations and Improvements

A One feature that the Nameserver implementation is lacking is that services are not automatically deregistered when the corresponding dispatcher exits. For that, a deregistration request should be sent the the Nameserver for every services that was registered from the exiting dispatcher.

B In theory it would be possible to extend the Nameserver to not set up a UMP channel for every lookup. Instead, it could set up an LMP channel if the service is running on the same core as the requesting dispatcher, which could increase performance. The tricky part there would be to put the LMP channel (which will comprise of an endpoint capability) into a response from the Nameserver (which has to be a UMP message). Since our UMP channels cannot transmit capabilities other than frame capabilities, this is currently not directly possible. Maybe this could be implemented using a local task that the Nameserver sends to the monitor running on that core, where the monitor sends the endpoint capability to the requesting process.

9. Shell

> *From a programmer's point of view, the user is a peripheral that types when you issue a read request.*
>
> — P. Williams

Individual Milestone: Andrin Bertschi

A shell is an essential part of an operating system. It allows programs to interact with the user and vis-versa. In this chapter, we dive into design and implementation of AOSH, the AOSH Operating System sHell. In order to implement a shell, we first detail the subsystems implemented to read and write characters from a serial interface. This is why this chapter is split into two main section. First, we set out how we run the UART driver in userspace and provide read and write functionality across cores. We then show an application of these APIs by implementing a shell. The shell comes with built-in commands and the ability to spawn domains on the system.

```
.............................
.......____...___..____..__..__....
....../ \  / _ \/ ___|| |.| |...
...../ _ \| |.| \___ \| |.| |...
..../ ___ \ |.| |...) |  _  |...
.../_/...\_\___/|____/|_|.|_|...
.............................
Welcome to AOSH..............
AOSH Operating System Shell.....
.............................

aosh >>> echo AOSH rocks!
AOSH rocks!
```

Listing 9: Greet dialog of AOSH

9.1. UART Driver in Userspace

When we started implementing the individual milestones we agreed that ultimately, most RPC services should run in their own domain. This requires a stable nameservice to register and discover RPCs. In the beginning, such nameservice was not yet implemented which is why the initial implementation of this milestone run within init/0. The steps to

access serial I/O in userspace were described in the handout and showed to be straight forward. In a first step, we initialized the lpuart3 driver which was already provided. We could then read and write to the serial port but relied on polling for new keys. The next step involved setting up the ARM Generic Interrupt Controller to get notified with character IRQs. This proved to be a lot more error prone as the next section details.

9.1.1. Interrupts not Delivered

We had difficulties with receiving interrupts. Some interrupts arrived but if too many keys were pressed, they got stock. It turned out that we did busy waiting in `init/0` in our URPC event framework (section 5.4.2) and did not yield on retry. The serial server run first within init/0 and was later on migrated into its own dispatcher. Busy looping had impact on delivery of IRQ. Introducing yields in the event loop generally had positive impact on overall system performance. But it was not the main cause of the issue. We struggled with missing interrupts throughout the milestone. After a while of using the shell, interrupts would suddenly stop to arrive. This made the shell unusable for testing of other milestones. A temporary fix included a mechanism to disable IRQ and use polling instead. This allowed the shell to be functional and helped the other team members to test their functionalities. It later turned out that the buffer of the UART driver turned full. Reading up on the i.MX 8QXP Applications Processor Reference as well as opening a moodle discussion helped to figure out that the whole data buffer needed to be emptied. The UART only sends an interrupt when one of the relevant status flags transition to 1. So if the buffer is not cleared, new interrupts may not be delivered anymore. Consuming all characters in the buffer on any given IRQ solved the bug and we re-enabled the IRQ delivery mechanism.

9.2. Implementing a Serialserver

With a functional driver in userspace, the RPC calls to read and write to serial I/O can be re-implemented through the UART interface.

9.2.1. Multiplexing Input

There are various ways to implement how serial I/O is multiplexed among domains. The most trivial implementation serves a single character to whoever is currently requesting. This comes with the disadvantage that a domain may not consume everything a user types. For instance, if we spawn two shells and both ask for input, both may get fractions of what the user types. This is clearly not desirable. An other idea is to acquire the serial I/O exclusively until it is explicitly closed by the domain (either through an explicit call or if a domain dies). This comes with the drawback that we have to introduce new RPC and we may break API compatibility with the existing code (as well as the grading functions). A middle ground between these two ideas is to de-multiplex input on a line basis. A domain can read until a new-line character arrives. This implies session management between server and client.

9.2.2. Session Management

The design of such sessions is inspired by Cookies in HTTP. The serialserver hands out new tokens if a request does not include already one. The client then stores the token and includes it in each forth coming request. The token identifies the client and dictates what has been read. The serialserver is either in an acquired state or a non-acquired state. For a newly acquired state, a ring buffer is allocated. This allows a client to acquire the serialserver, read from input but not fully consume all typed characters. As soon as the user hits return[1], the state of the serialserver goes into non-acquired state. With this design, not yet fully consumed input can then still be consumed even if the request does not own the serialserver exclusively anymore. If all input is consumed, the ring buffer associated with token of the client is freed. Diagram 9.1 depicts the high level idea of the session management.

[1] ASCII NL, 10), carriage return (CR, 13), or end-of-file (EOT, 4)

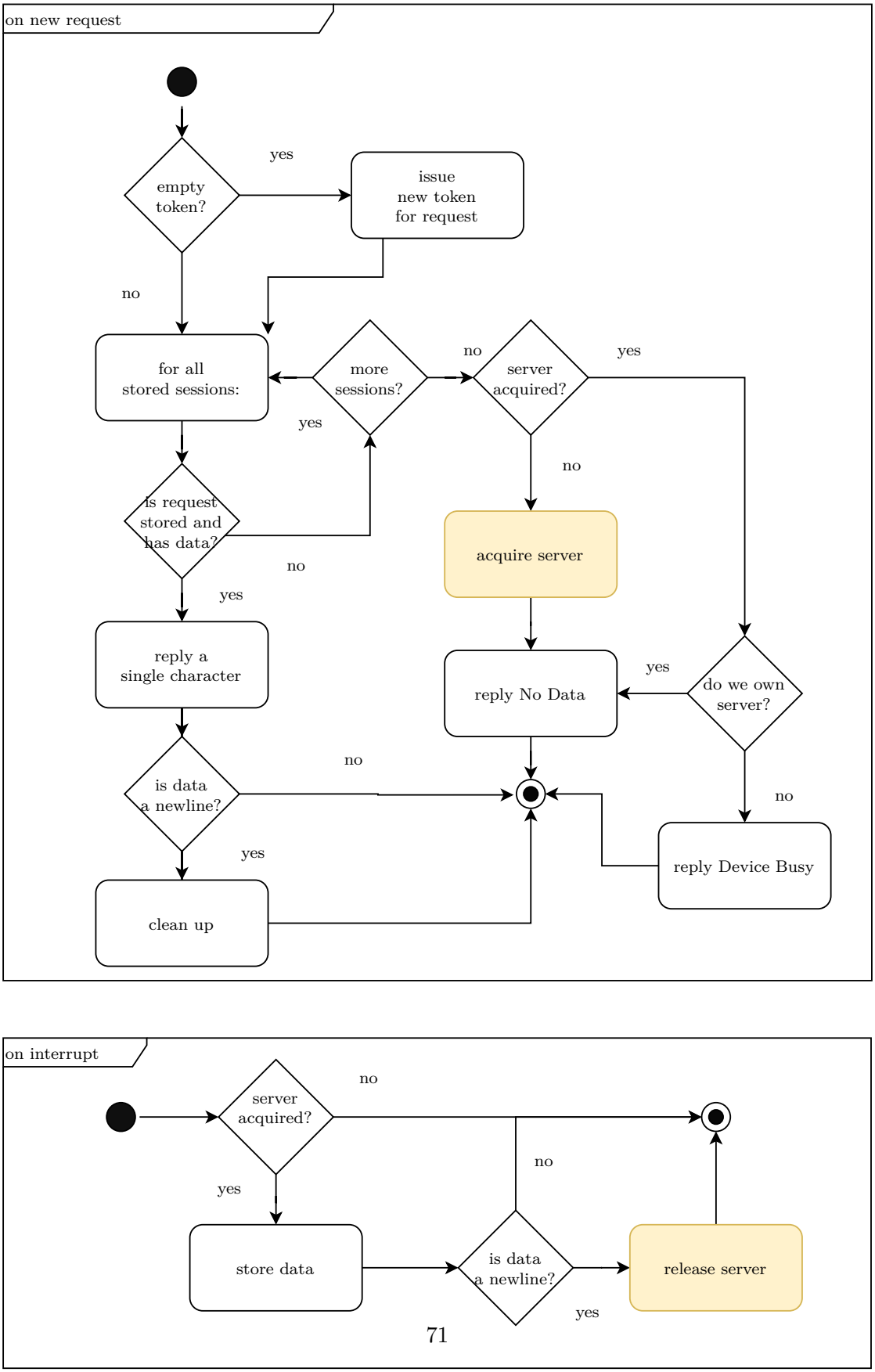

Figure 9.1.: Activity Diagram of Session Management. Yellow are changes to the acquired state of the serialserver.

Limitations and Implications

First come/ First serve The serialserver is acquired on a first come first serve basis. For further improvement, this can be implemented in a round robin fashion.

Session Tear Down A client may acquire the server and only read a single character. There is currently no feature implemented to tear down outdated sessions. We can think of a periodic cleaning event which removes sessions if they are kept online but never fully consumed. Alternatively, an explicit RPC call in tear down logic of an exiting domain releases a not fully consumed session.

Ring buffer for Sessions The list of clients which have acquired the serialserver is implemented in a linked list. This implies that lookup of previously acquired but not yet fully consumed sessions is linear in the number of such sessions. For tighter memory constraints, the linked list can be implemented in a ring buffer itself. This then allows a limited number of not yet fully consumed sessions. We left such optimizations open for further improvement.

Server calls Client The design of the session management originates from the time during development when no name service API was available. There is currently no support for the server to call the client such that client does not have to poll. The name service added support for such feature. This was also implemented in the networking stack to de-multiplex incoming packages to the correct domain (section 11.4). This approach would scale better in the number of clients asking for serial input. The polling design did not have noticeable impact on the system and serial I/O performed smoothly. Under the time constraints and pending bugs in the subsystems, this feature received low priority and was left as an enhancement. If clients do not acquire the serialserver and do not get data, they retry again later on.

9.2.3. Migrating to Nameservice

By the time the sessions management was implemented, the name service functionality gained stability. This allowed us to migrate the serial server from `init/0` into its own dispatcher. Apart from transferring capabilities to device registers and IRQ destinations, this tasks mostly included testing whether the same functionality is stable on the name server. It required new marshalling logic in the `aos_rpc` wrappers as well as de-multiplexing of incoming `rpc_messages`. The name service APIs were mostly backwards compatibility to how we served URPC requests already. However, integrating subsystems while still being in early development should not be underestimated. By the time we got closer to submission date we increased the frequency of meetings and solved questions regarding integration bytes at a time. See the chapter of the name server for more details.

9.3. AOSH Operating System Shell

This section descries AOSH, the AOSH Operating System Shell. AOSH implements a simple read-evaluate-print loop (REPL). Input characters are consumed through the serial server, parsed into commands and arguments and then dispatched to the plugin system. Since all individual milestones will write shell built-ins, it was a requirement to introduce a plugin system to easily add new built-ins. Built-ins can be sourced in their own c-modules and are added to a collection. AOSH then queries the correct built-in based on its name and invokes its function pointer.

```
struct aosh_builtin_descr aosh_builtins[] = {
        {builtin_help,     "help",     "prints this help"},
        {builtin_time,     "time",     "time a command"},
        {builtin_oncore,   "oncore",   "spawn a dispatcher on a given core"},
        {builtin_ip,       "ip",       "show IP address"},
        ...
```

Listing 10: Simple dispatch system for built-ins

For a list of all implemented built-ins, the reader may consider the User Guide in appendix B.3.

9.3.1. Spawning in Foreground

While implementing the builtin `oncore`, which spawns a domain on a given core, we noticed that the shell does not have a concept of what is currently in foreground. One may want to spawn a dispatcher and then not be interrupted by the shell which continues with its REPL. This is why we introduced a concept of foreground activity. With the `-f` flag, `oncore` will spawn a new domain in foreground. There are different designs how to implement such activity. The process server needs a way to know which of its processes are currently running.

- The process server may frequently ping domains to see if they are active and have not crashed. Every dispatcher then has to understand these messages and reply accordingly.
- During tear down, a domain tells the process server that it is about to finish. This removes pinging traffic. However, this does not hint whether something is responding well or is stuck. This design needs to solve authentication. If a domain can signalize its death, how can it be prevented that it signalizes death of another domain?

To minimize congestion of RPC messages on the system, we opted for the second design. Implementation includes new RPC calls to signalize death, as well as querying status

information. We postponed the authentication issue; Signalizing death of someone else does not have implications on its runtime. It merely updates status information. It could be solved by issuing tokens or capabilities. This, however, was not further considered. With this functionality the shell now can spawn a new dispatcher, and query its status information until that domain has exited. This allows to spawn domains in foreground and resume the shell on exit.

9.3.2. Command History and Cursor Navigation

Implementing readline functionality on our own seemed to be tedious and we reckoned it may be less work to port an existing library. It turned out Linenoise, a BSD licensed, self-contained alternative to readline and libedit[2], was perfect for this job. Porting it was done in a few ours. We do not have POSIX compatibility which is why some features had to be disabled and some functions calls had to be redirected to our RPC services. AOSH then supported line editing, cursor navigation and history features.

9.3.3. RPC call Putstr instead of Putchar

A new RPC call was added to print a string and not only a single character. Libc handles already included a length so this could easily be forwarded to our RPCs. This changes reduces the number of RPCs done to send a string and therefore leads to less RPC congestion.

9.4. Optimizations and Improvements

Some ideas for optimizations were already mentioned previously in section 9.2.2.

Round Robin Serving The submitted code contains dispatch of serial sessions on a first come/ first serve basis. Alternatively, an extension was started to be implemented to serve in a round robin fashion. This would make acquiring sessions more fair and was originally planned to be shipped. Due to time constraints and higher priority to-dos on the code base, it did not make it in the submission.

Prevent Session Hijacking The session is currently represented as an integer which is incremented for each new client. In order to prevent session hijacking, the session size should be made sufficiently large and be a random number issued by the server. In was reasonable within the scope of this course to leave the session vulnerable for hijacking. For a production ready deployment, this however, is a vulnerability.

Output Inter-leavings with CPU-Driver In our system, the serial server in userspace, as well as both CPU-Drivers access serial output. The latter ones through `SYSCALL_PRINT`. If debug output is enabled and CPU-Drivers as well as serialserver print output, one may seem inter-leavings on the screen. This is however not a bug but simply stems from the fact that CPU Drivers as well as the serialserver access

[2] https://github.com/antirez/linenoise

the driver. In a production ready system, `debug_printf` output may be written to a file or the serial server may be used more extensively in other applications.

10. Filesystem

> *All problems in computer science can be solved by another level of indirection.*
>
> — David Wheeler

Individual Milestone: Loris Reiff

In this chapter introduces the file system with rudimentary FAT32 support. It covers the different design decisions and difficulties of integrating it into the system. Furthermore performance evaluation of the implementation are presented.

10.1. Block Driver

A block driver for the SDHC reader of the board was already provided. The interface was straightforward and allowed to quickly implement a block driver server on the old infrastructure prior the nameserver chapter 8. Having a block driver server is a sensible approach as it provides a strate forward way to serialize access to the block driver. One might wonder why there is even a section about the block driver. Well, it turned out that there were some substantial issues when using the block driver with our URPC framework. The block driver worked splendidly *on its own*. Our URPC message passing did its job great *on its own*. However, combining those two lead to some mysterious issue. The DMA of the block driver seemed to fail and the results could only be read if some delay was inserted. However, no errors were returned and everything seemed fine, except that nothing was present in the buffer and the `sdhc_test` didn't pass. After some debugging and some discussion with the team, removing a `thread_yield` at a blocking rpc receive seemed to fix the issue. This resulted in a bit more sluggish system. A scarifies that was reasonable, as the rpc framework was being remodeled anyway for the nameserver and the clock was ticking. On a downside, the origin of the problem was never found and we may die without knowing the cause of the issue...

10.2. FAT32

With the block driver running as a service that serializes read and writes, the next logical step was to implement FAT32 support. Read and write requests are obviously passed to the block driver server. But before diving into the design decisions, which are covered in section 10.3, let's recap some of the basics of FAT32. FAT is a filesystem introduced by Microsoft and its origins date back to the 1970. There are different variants of it (e.g. FAT12, FAT16, FAT32). The basic design is the so called file allocation table (FAT). This

is a linear array of blocks as depicted in figure 10.1. Each entry marks a corresponding cluster on the volume as free or used. If a cluster is used the corresponding FAT entry points to the next cluster that contains the following data. Directories are special files and its entries will contain a cluster number as well, with which one finds the first cluster.

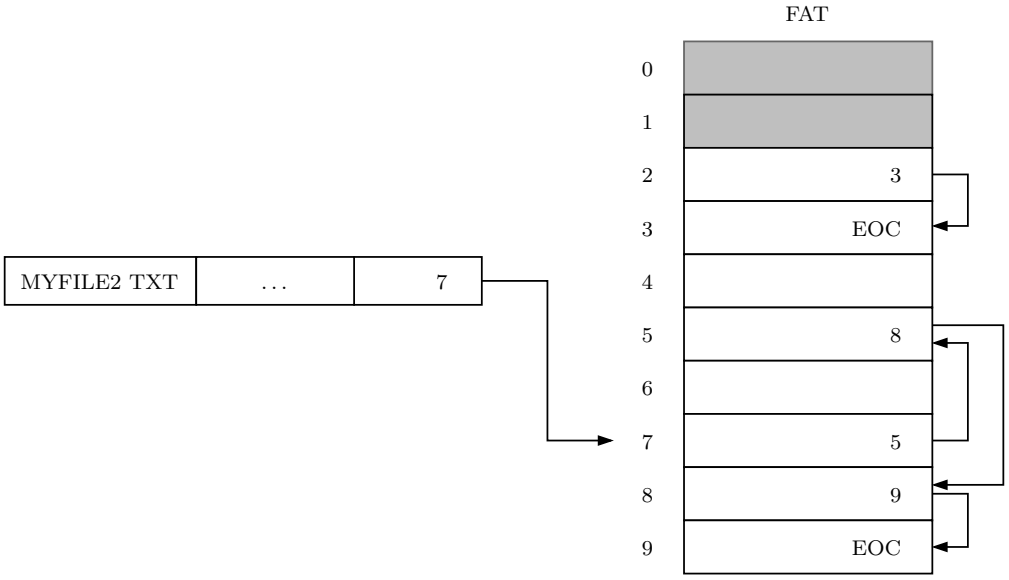

Figure 10.1.: The File Allocation Table (FAT)

10.2.1. On Reading Specifications Carefully

At one point when the FAT32 implementation was stable, a weird artefact occurred. When a directory was created on the our barrelfish fork, and the file system was then mounted on a Linux Fedora machine, non-existing directories were shown. A `ls` on our system did not reveal this directory. After examining the SD card contents in a hexeditor, everything seemed to be correct. So the specification were reconsulted again, it turned out that an optimization in our implementation led to a discrepancy between our implementation and the specification. The specification states the following:

> If `DIR_Name[0] == 0x00`, then the directory entry is free (same as for `0xE5`), and there are no allocated directory entries after this one (all of the `DIR_Name[0]` bytes in all of the entries after this one are also set to 0).
>
> p. 23 of FAT32 File System Specification

In order to safe on writes only the first sector of a cluster was zeroed when creating a new directory, as `DIR_Name[0] == 0x00` indicates that there are no allocated entries afterwards. However, the text in the parenthesis also states that all following entries should start with a zero byte. Which was happily ignored while reading the specifications

the first time. It was therefore also not reflected in our implementation. This was then fixed.

10.2.2. Limitations

The following limitations are present in our FAT32 implementation:

- Only support for 512bytes sectors, this should be fairly easy to adapt
- No support for timestamps
- No long file names support
- FSInfo specific values like for instance `FSI_Nxt_Free` are not updated

10.3. Integration

To be able to test the FAT32 implementation while development, the necessary parts were directly integrated in the libc abstraction in a vertical manner. That is as soon as the necessary subroutines to create a directory in FAT32 existed, it was glued with libc. That is in a first iteration the filesystem was used as a library. This comes with its downsides as we see in a minute. However, this approach allowed quick progress and didn't lead to a lot of dependency on for instance the nameserver which was developed in parallel.
As soon as the nameserver API was stable the block driver server was migrated to it. Luckily the SD reader issue described at the beginning of this chapter was resolved with the new message passing infrastructure, without resulting in a slow mess. At this point the limitations of the simple filesystem library must be fixed. The block driver supports a atomic read and an atomic write. The read and write requests are being serialized by the rpc framework. However, if each dispatcher is running its own instance of the file system The file system might end up in an inconsistent state as it has to read a block and update it (see figure 10.2). There are of course different solutions to this issue. One is to have an atomic update functionality on a finer granularity. Or state could be introduced, a dispatcher could for instance block the block driver and other dispatchers sending request would have to wait. Or one could introduce a filesystem server. Let's take some time to reflect upon the different implications of a filesystem as a library vs a server.

FS as a library As already mentioned if we design the filesystem as a library we need more fine grained access to serialize operations on the block driver. Another aspect to consider is how to share a reference to a file between dispatchers. For instance if one would like to introduce the ability to share file descriptors between dispatchers. With a file system as a library one would have to share quite a bit of data depending on the underlying file system. Furthermore one must decide what happens when different dispatchers want to access the same file. Does this

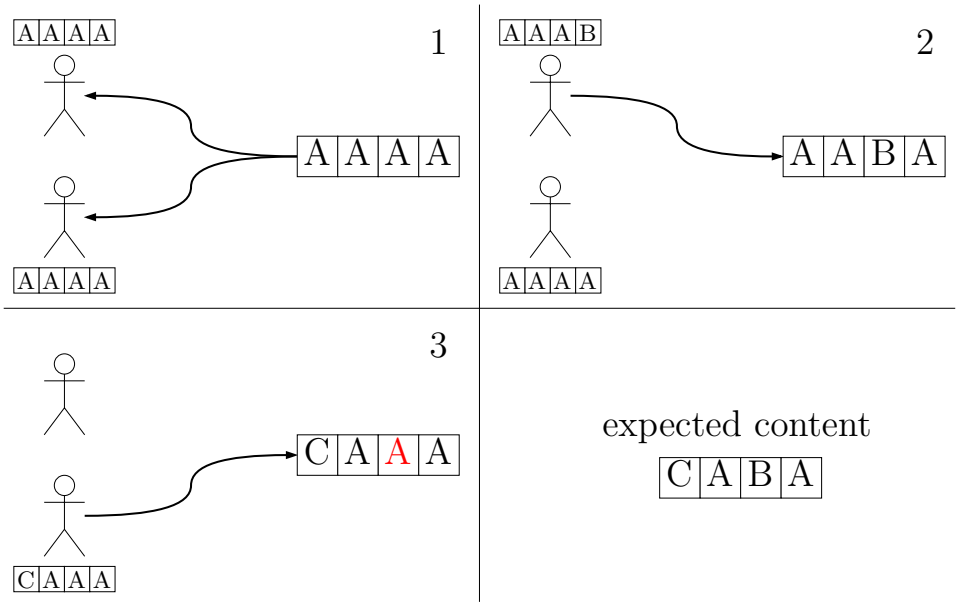

Figure 10.2.: Inconsistency with only atomic read and write. Two parties read a block perform modifications are write back the block.

involve some synchronization between the library? This seems to be a nontrivial task. Lastly, how can we include access control into a library? Maybe one wants to limit write access to certain sub-directories. This can not be provided on a block driver basis.

FS as a service If there is only one file system service that accesses the block driver, we can get away with atomic read and writes on a block basis. Sharing of a reference to a file can also be done easily. The file system service can keep local state and share a identifier (or a dedicated capability) with the parties opening files through it. Also the case where multiple dispatchers access the same file can be handled gracefully without too much thought. One idea is for instance to implement reference counting and prohibit the deletion of an open file. Also fine grained access control is easier when a filesystem server is present.

Looking at the different implications, it seemed clear that designing a feature rich and easily extensible file system appears to be easier with a dedicated service. That is why we opted for this approach. An option we didn't touch upon is to integrate the file system server with the block driver server. This breaks some abstraction and might be desirable if the rpc calls turn out to be a performance blocker. It follows a brief description of the newly added API and the implemented design.

10.3.1. Block Driver Server

The block driver service basically provides a wrapper service for the block driver. There is a call to read a block and on to write a block. Even though the calls have parameters for the block size there are restricted to 512 bytes. This should allow for easy extensibility if larger block size was to be added.

```
struct aos_rpc *aos_rpc_get_block_driver_channel(void);

errval_t aos_rpc_block_driver_read_block(
    struct aos_rpc *rpc,
    uint32_t index,
    void *buf,
    size_t block_size
);
errval_t aos_rpc_block_driver_write_block(
    struct aos_rpc *rpc,
    uint32_t index,
    void *buf,
    size_t block_size
);
```

Listing 11: API for block driver server.

10.3.2. Filesystem Server

The filesystem server provides abstract filesystem operations such as opening, writing, reading, etc. Three rpc calls can be seen in listing 12 and the full set of operations can be found in `include/aos/aos_rpc.h`. The interface is designed in such a way that it can be easily be used by libc wrappers. Some notable limitations are that the filesystem server currently does not provide any sort of reference counting or other means of bookkeeping of which files are open. That is a dispatcher could delete an open file which results in undefined behavior. This functionality could be added easily. The sole reason why it wasn't added were issues while implementing the spawning functionality which took quite some time. For our use-case this isn't a big issue though. Another potential issue might be that the filesystemserver returns a pointer used in its virtual address range. A malicious user could potentially use this information to facilitate exploitation. This could be fixed by having another layer of abstraction.

10.3.3. Libc Integration

The heavy lifting of the filesystem work is done on the file system server and the gluing to libc was straightforward with the provided abstraction in barrelfish. Here is a non-extensive list of supported functions:

```
errval_t aos_rpc_fs_open(
    struct aos_rpc *rpc,
    const char *name,
    file_handle_t *handle
);
errval_t aos_rpc_fs_close(
    struct aos_rpc *rpc,
    file_handle_t handle
);
errval_t aos_rpc_fs_write(
    struct aos_rpc *rpc,
    file_handle_t handler,
    char *buf,
    size_t size,
    size_t *written
);
```

Listing 12: API for the filesystem server.

- `fopen`
- `fclose`
- `fread`
- `ftell`
- `fseek`
- `readdir`
- `fgetc`
- `fwrite`
- `mkdir`
- `rmdir`
- `rm`
- `fstat`

10.4. Shell Built-ins

We decided to develop `cat`, `ls` and friends as shell built-ins as spawning happens over a dedicated command (see listing 13 for a complete list of supported built-ins). The development was straightforward as the standard libc calls can be used for file operations and the built-in interface is easily extensible (section 9.3).

10.5. Spawning

At this stage our spawning implementation was quite clean and adding the functionality to spawn from the filesystem seemed straightforward. However, limitations of the nameserver lookup (covert in section 8.2) made the whole procedure more involved. The underlying issue is that the `init` dispatcher can not perform nameserver lookups. So the first idea was to send the filesystem server endpoint to monitorserver which runs on `init` and handles the spawns, which could then query the filesystem server. However, this turned out to be impossible as well, which we could have realized before implementing. The issue

is that the filesystem server might request more ram, while the monitorserver running on `init` dispatcher waits for an answer from the filesystem. During that time the `init` dispatcher can not reply to requests. And since the memory server runs on the `init` dispatcher the filesystem will never receive an answer, i.e. we are in a deadlock situation. This approach had therefore to be scratched. We instead opted to read the binary from the processserver and then transfer the binary data via rpc as depicted in figure 10.3. As

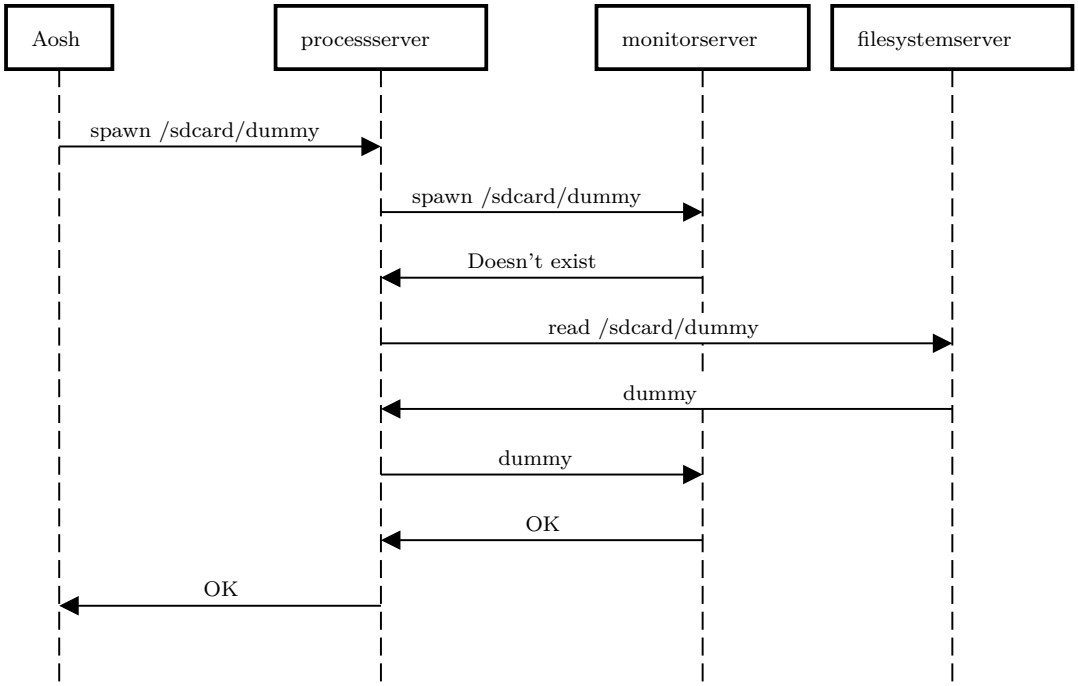

Figure 10.3.: Spawning an executable from the filesystem

binaries can get quite large our way of sending larger rpc messages is not ideal, as this will cause some performance penalty. However, this is not the bottleneck here as we will see in the next section where we evaluate the performance.

10.6. Performance Evaluation

A performance evaluation is very much needed indeed. Spawning takes over 10 minutes. So what is the culprit? Evaluating the block driver reveals the bottleneck. If we take a look at figure 10.4 we see that both writing and reading a 512bytes block takes almost half a second on average. Note: this does not include any rpc messaging! This is a measurement of 200 reads and 200 writes. Almost a third of the time is spend waiting for the device. We ran rudimentary tests on a modern laptop and the SD card seems to support a read speed of around 20 MB/s. So this indicates which component has to be optimized the first! For additional overhead that is introduced through rpc overhead, consider the measurements in the dedicated section: section 6.4

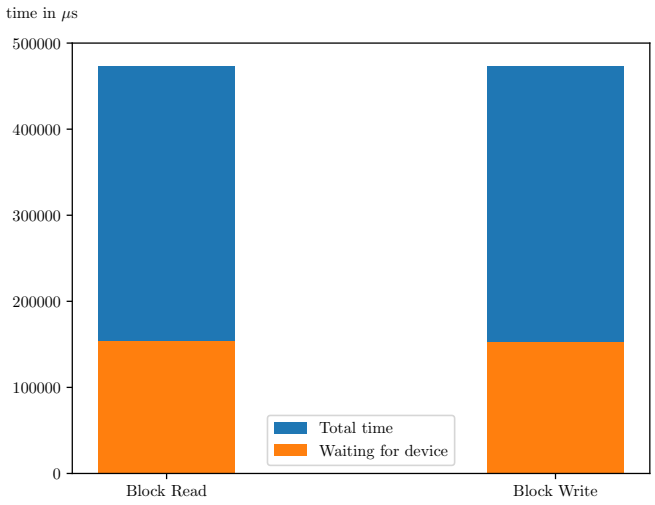

Figure 10.4.: Block driver

The peak writing performance we achieved after freshly formatting the SD card was at 533.34 B/s and the maximal reading bandwidth was 1066.84 B/s. Figure 10.5 shows average read and write time. However, the performance is highly dependant on the fragmentation as the FAT has to be considered. That is why we measured with a freshly formatted SD card. The test can be found in our repository in the application `fs-bench`.

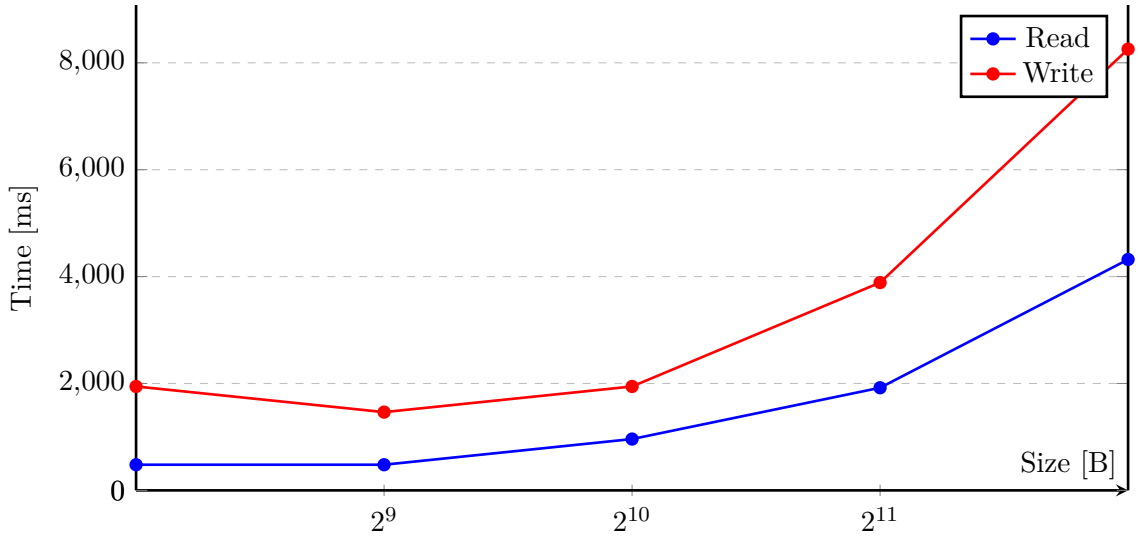

Figure 10.5.: The average of 20 reads and writes after formatting.

10.7. Optimizations and Improvements

There are still plenty of places where optimizations, improvements and extensions can be added. This list is incomplete, since there are many file systems that could be for instance added ;)

Block driver speed The block driver is extremely slow as one can see in the performance evaluation. It is the first part that should be improved to get better performance.

Caching At the moment the block driver service performs all reads and writes directly from the SD card. It would be more sensible to cache the reads and writes in memory and only flush them on a dedicated flush call.

Fix limitations of FAT32 The FAT32 implementation is not complete as mentioned in section 10.2.2. Those points can be addressed

Access control At the moment there is no form of access control to the file system or block driver service in place. This holds true for all services of our system. However, ideally one could limit access to services or sub-directories.

Sharing of file descriptors The design with the file system server makes it easy to share file descriptors. The rpc API would have to be extended to share the underlying `fdtab_entry` struct.

Keeping track of open files The file system server currently does not perform any sort of reference counting or similar. Therefore an open file can be deleted which leads to undefined behavior.

Mounting Currently the SD card is mounted while booting, it is not possible to mount it afterwards. Support for mounting and unmounting can be added.

11. Networking

> *However, as every parent of a small child knows, converting a large object into small fragments is considerably easier than the reverse process.*
> — Andrew S. Tanenbaum

Individual Milestone: Raphael Eikenberg

In this chapter, the design and implementation of a simple network stack is described. There were multiple challenges to fulfill the requirements of the milestone. The most important steps were

- initializing the network device,
- implementing the different network layers as the core of the stack,
- multiplexing UDP connections and integrating the network service with the system-wide RPC mechanism, and
- benchmarking the performance of the network stack.

The steps were approached in this exact order, and will be explained in more detail in the following.

11.1. Initializing the Network Device

Before we can communicate with the device, we need to initialize it, and hand over some memory to setup the device queues.

11.1.1. Mapping the Device Registers

In order to be able to communicate with the network device, we first need to acquire access to its registers. Using a call to `map_driver()`, the device registers are mapped into our virtual memory.
Due to a bug in the handout, the frame needs to be mapped non-cachable. Otherwise, data written using the device queue will not be correctly read by the device. Passing `VREGION_FLAGS_READ_WRITE_NOCACHE` to the mapping call results in correct behavior.

11.1.2. Hardware Configuration and Device Queues

The handout included a device queue layer for the device, a Mackerel specification for the device, and some initialization code. The provided function `enet_init()` takes care of the hardware setup.
To receive packets, we need to feed some memory in form of buffers to the device. When the hardware receives data, it stores it inside these buffers. Once we query the device queue, one of these filled buffers is handed over to us.
We equip the device with 512 buffers of size 2048. Note that not the full size is needed to transfer the contents of the Ethernet packets. First, we have to allocate memory for those buffers, after which we register the memory using `devq_register()` as required by Barrelfish's API for device queues. A call to `devq_enqueue()` will transfer the ownership of the buffers to the device.
Equivalently, we register some memory for sending Ethernet packets. In contrast to the procedure from above, the buffers will not be enqueued by default—we maintain ownership until we actually send a packet.

11.2. Sequential Processing of Packets

Now that we have device queues, one may ask how exactly an Ethernet packet is received. During initialization, we set up a periodic event that is called with a 20 µs delay.[1] Using that event, the driver continuously polls the receive device queue for new packets.
The whole driver works in a sequential fashion: it processes one buffer after another. This reduces complexity, and results in the fact that at maximum a single receive buffer is owned by the driver at a time—the hardware owns all the other buffers. The reason for this is that after processing a buffer, we hand the buffer back before pulling the next one. An equivalent invariant holds the send device queue. When a buffer is handed over to the device, we wait until the buffer is released again.
This design decision comes with its up and downs. On the one hand, the design is simple and straightforward. Networks and protocols are already complex enough, so we should not add complexity where it is not needed. On the other hand, the restriction makes it more difficult implement an "active" network device, i.e., a device that can initiate communication on its own. Due to the sequential nature of our driver, an active device may be required to wait for a specific response to a request that we sent. To wait for the response, we could

- store the current state and process it later while serving the event loop,
- poll the receive queue, store the content elsewhere, and directly hand the buffers back, or
- poll the receive queue, and acquire ownership of all buffers until our response was received.

[1] As will be shown later in the chapter, this delay yields reasonable performance.

The third option is not possible with the current design, which means we *have to* store either some state or the direct buffer contents outside of the queues.
Apart from that, it should be pointed out that the third option has a flaw anyways. If the response is not among the first 512 buffers, we have to drop the currently processed packet, because otherwise we will lose data.

11.3. Network Layers

For designing the different network layers, it made sense to stick to separation given by the OSI model. In our basic network stack, we will cover the protocols

- Ethernet and ARP,
- IP and ICMP, and
- UDP.

A structured overview of the protocols supported by the stack is illustrated in figure 11.1.

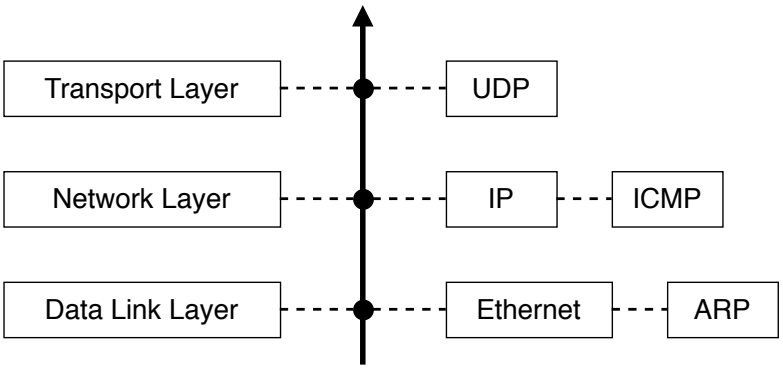

Figure 11.1.: The supported protocols and their relation the the OSI model.

In this implementation, each protocol is more or less strictly contained in its own module. However, some modules depend on other modules. For instance, the `ethernet` module depends on the `arp` and `ip` modules, because upon reception of a packet, those modules could be interested in further processing the packet. The dependencies are depicted in figure 11.2.
This section will highlight the Ethernet and ARP implementations, where most time was spent during implementation. The other modules followed the same patterns.

11.3.1. General Module Structure

Each protocol module implements a processing function, which takes the packet, determines if it can process the packet, and either passes it on to the next layer, or calls some callback of the driver.

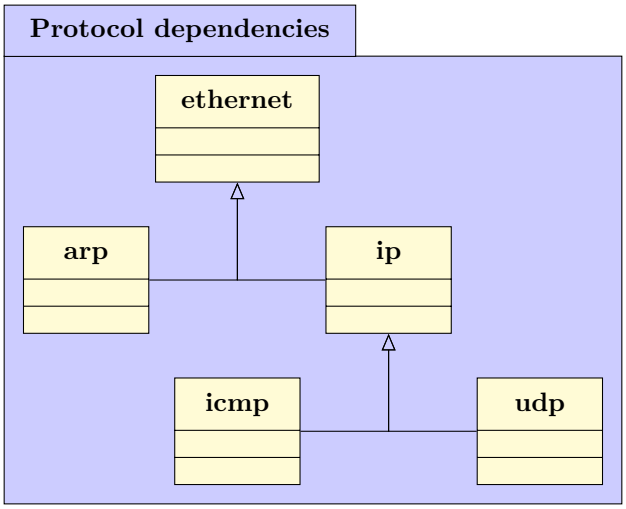

Figure 11.2.: Some protocol modules depend on other protocols.

Further, most protocols provide a function for sending packets. As an example, when the IP implementation passes data to the Ethernet implementation, the data will be wrapped and a header is prepended.

11.3.2. Ethernet

The most fundamental protocol for the network stack is Ethernet. For transmitting a packet on any of the other protocol we will discuss in this chapter, Ethernet will always be involved as the outermost-layer.
On initialization, the MAC address of the device is read. Afterwards, the modules for ARP and IP are initialized.
The board communicates with the static IP address 10.0.0.2. As later shown in the user guide, this address can be shown using a shell builtin.
All received packets are passed down to the `ethernet` module. To accept a packet, it must either

- be addressed to our device, i.e., the MAC address of the receiver must match the one we read from the device, or
- be sent as a broadcast (so that all bits in the MAC address are set).

Packets not fulfilling this requirement are dropped.
Depending on the type, the packets are then passed to the modules for ARP or IP.

11.3.3. ARP

ARP is an essential part of a network environment, and is especially useful for debugging. With the ARP protocol, the network stack can acquire knowledge of the corresponding MAC address for a given IP address.

For efficiency, ARP queries are cached in a hashtable with 256 buckets.
The ARP cache can be requested by other dispatchers over RPC. An example of that is shown in the user guide.

11.4. Multiplexing UDP Connections

The driver domain provides network functionality to the rest of the system. This is done via RPC calls.
During initialization, the driver dispatcher needs to register the networking service at the nameserver. The name of the service is `networking`.
Using the nameservice library, other dispatchers can utilize the networking stack for their own tasks. To make communication with the networking service easier, a library is provided in `include/aos/networking.h`. Dispatchers can provide a callback when registering for a port. The callback is called when a new packet for the registered port is received.
Internally, the driver registers new "bindings", and inserts them into a hashtable. As a key for the hashtable the port seems to be a reasonable choice. That way the mapping can be easily queried once a new datagram is received.

11.5. Performance Evaluation

Of course, networking is only half the fun when it does not satisfy the requirements in speed of the user. In this section, we will carefully analyze different performance aspects of the networking stack.
The measurements were performed at the very end of the milestone, and should be consistent with the submitted version of the code, i.e., all claims should be verifiable using the submitted code.
When a measurement involved using the `echoserver`, the debugging output–printing the datagram payload–was disabled.
Overall, the performance of the network stack suffered from activity on the operating system. Once too many processes are spawned, the network stack slows down in both bandwidth and latency. For this reason, in the following measurements, only the default services were spawned, and a single `echoserver` or `udpsender` instance was running.
Additionally, the original design of the generalized UMP server lead to bad performance. Originally, we processed only a single client when serving the server. The performance analysis helped us spot this weakness, so that we decided to process all clients at a time.

11.5.1. ICMP

For ICMP, we are primarily interested in the latency. There is not a lot of data usually transmitted via ICMP, thus we skip the bandwidth analysis here.
The measurements were performed with `nping`. As a delay between echo requests, 0.2 s was chosen.

Figure 11.3 shows the response time for echo requests as a boxplot.

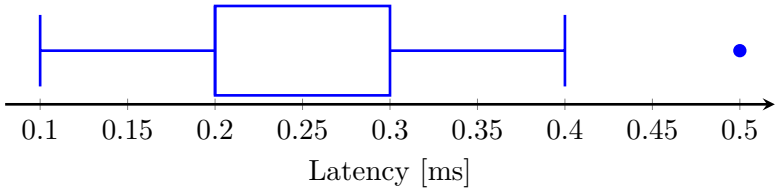

Figure 11.3.: Latency of ICMP echo responses at a sample size of 256.

11.5.2. UDP

For UDP, both latency and throughput were deemed relevant. Both properties were measured individually.

Latency

Similar to the ICMP measurements, `nping` was instrumented. Due to its flexibility, `nping` could be used without modification to measure the time of the UDP round trip. The delay between echo requests remains unchanged.
Figure 11.4 shows the response time for UDP requests as a boxplot. As expected, the latency is higher than for ICMP. This is due to the additional URPC exchange between network service and `echoserver`.

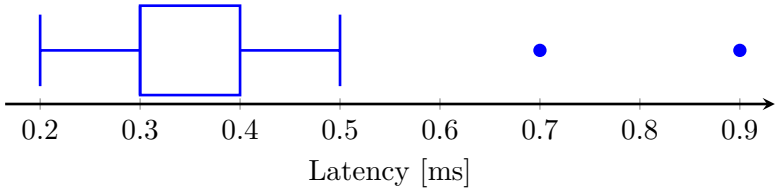

Figure 11.4.: Latency of UDP responses at a sample size of 256.

Throughput

Throughput can be measured in two ways. One might be interested to download data from the board, or upload data to it. For each case, a different setup was chosen to better suit the needs of the measurement.

Download For measuring the bandwidth for downloading data from the board to the PC, a new program was implemented. The program registers for a UDP port and waits for a packet. Once a packet is received, it replies with a large amount of UDP datagrams. The program is called `udpsender`, and takes three arguments. The arguments specify

- how much data each datagram should contain,

- how many datagrams should be sent, and
- what port to listen for.

Figure 11.5 shows the throughput with which data can be transmitted from the board to the computer. For each payload size, 100000 diagrams were transmitted.
Interestingly, although the payload size increases exponentially, we see a slowdown of the respective growth in the bandwidth. As expected, the peak bandwidth can be reached with the largest payload size.

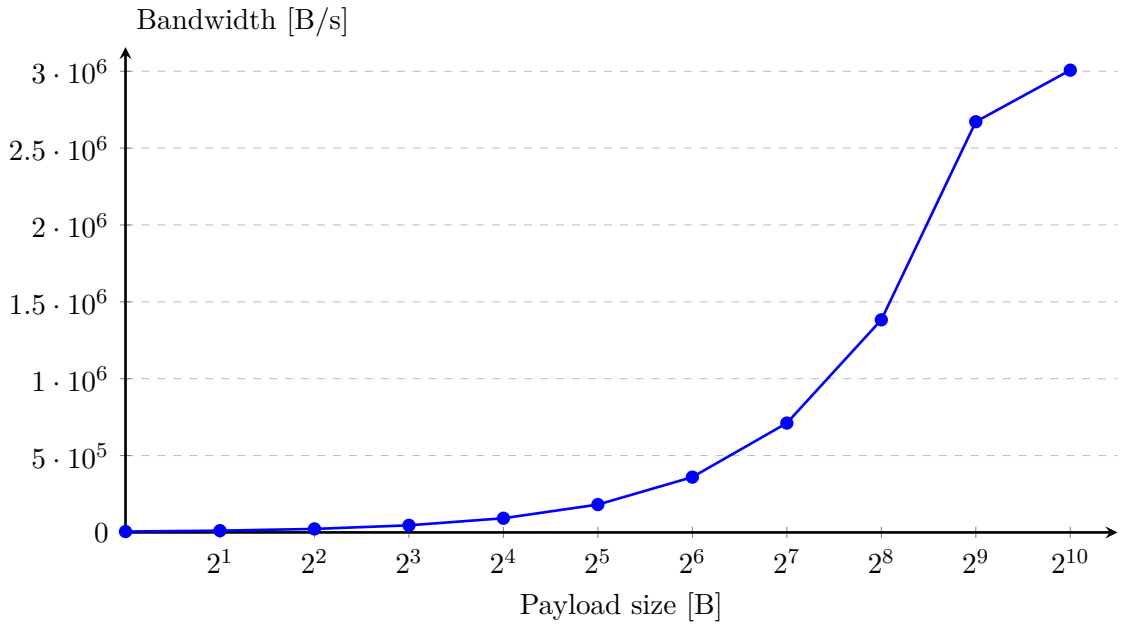

Figure 11.5.: Throughput when downloading data from the board to the PC for different payload sizes.

Upload Instead of measuring the bare upload throughput, we next wanted to see how fast we can "bombard" the `echoserver` program with datagrams. This gives us insight into how well the network stack can receive and respond to packets, i.e., its responsiveness.
What we measure is for how many datagrams we receive a response under a certain pressure. Here, the pressure is the rate with which we send data, in datagrams per second. From that, we can calculate the loss rate—how many datagrams get lost.
Figure 11.6 shows the throughput with which data can be transmitted from the board to the computer for a given pressure. The datagram count was adjusted in a way so that all measurement lasted for 10 s. That is, for a pressure of 100 datagrams per second, 1000 datagrams were sent.

As we can see, the loss rate jumps above 70 % with a pressure of a bit more than 500. This seems to be related to the device queues. If we do not process buffers in time, all 512 buffers are filled after some time. Once all buffers are full, we have to drop datagrams.

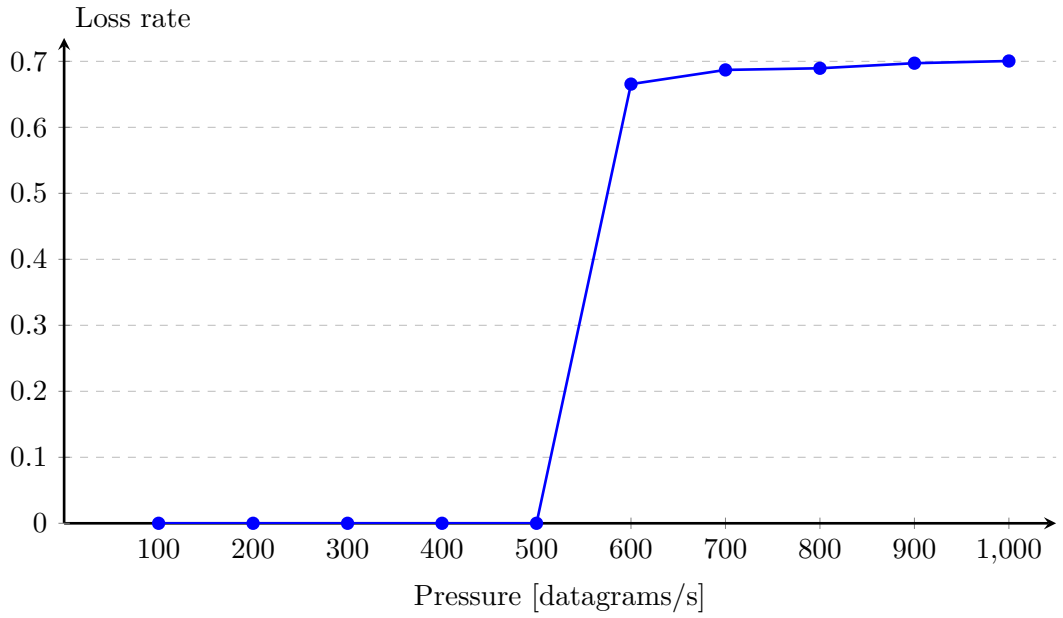

Figure 11.6.: Loss rate of echo datagrams by pressure.

11.6. Optimizations and Improvements

Cachable Device Registers Currently, the device registers are mapped non-cachable. This is a limitation inherited by the handout. It could be fixed by adjusting the barriers in the device queues of the Ethernet device.

Parallel Processing of Buffers With a stable threading implementation, the network buffer could benefit from more flexibility in processing different data streams at once. Of course, a mutex would be needed to synchronize access of shared data structures. In its current design, the network stack is very simple regarding control flow, which makes it easy to debug.

UDP Checksum Currently, the UDP checksum is not checked upon reception. Also, when sending a datagram, the checksum is not calculated. As per RFC 768, the checksum can optionally be set to zero.

Full Support for ARP Resolving The ARP implementation supports sending of ARP requests and reading responses. However, the UDP stack cannot make full use of it. When a program wants to send a UDP datagram, while the corresponding MAC address is not yet registered in the ARP cache, the sending will fail. However,

the ARP stack will notice the missing entry, and synchronously send an ARP request. That means, if the caller was to submit the payload a second time, the network stack could send the datagram in case the ARP response was received in the meanwhile.

This limitation results from not being able to defer sending of UDP datagrams. We could allocate memory to store the datagram until the ARP response was successfully received.

A. Development Methodology

For structured development, we agree on certain methodologies for our work. In this chapter, we cover how we managed to work on a large code base as a group.

A.1. Task Delegation

Throughout the whole project—except for the individual tasks—we have worked in pairs. In essence, our agreed-upon work philosophy was that we would be more efficient if we split the work into two main tasks, which are then approached by the two teams individually. This allowed us to make use of pair programming sessions.
Of course, we wanted to have two independent tasks, so that the pairs are flexible in how they organize their work. For some milestones, we were not able to split the given tasks, since they were dependent on another. Hence, sometimes we would come up with additional necessities, to make our system more stable or to ease implementing future features.

A.2. Communication

We maintained three communication channels.

- First, we set up a messaging channel. This was used to schedule team meetings, and served as the main channel for general problems and organizational questions.
- We used GitHub to discuss code and for keeping track of critical bugs. The platform turned out to be well-suited for feature-related discussions, since its integration with the code makes it easy to navigate. Figure A.1 shows an excerpt from our discussions on GitHub. In total, we submitted more than 120 pull requests and pushed more than 1200 commits.
- For integrating the work of our two pairs, we held meetings using a voice chat. This was especially helpful to see how the other pair is doing, and helping them in case some unexpected issues arose.

A.3. Continuous Integration

The lecture provided a GitLab repository, but we were not given the ability to open pull requests or setup continuous integration for it. Hence, we decided to work with

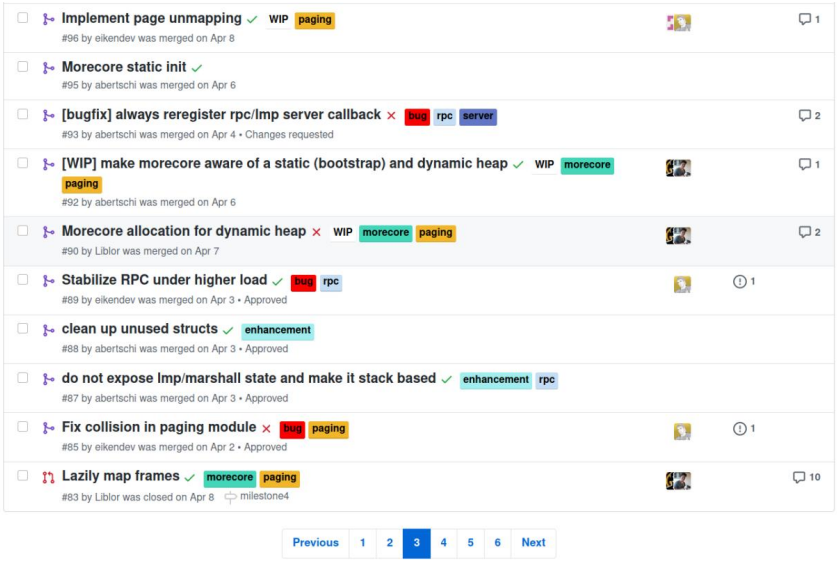

Figure A.1.: We mainly used GitHub for code-related activities.

GitHub, where we installed a configuration for Travis CI. Our architecture is depicted in figure A.2.

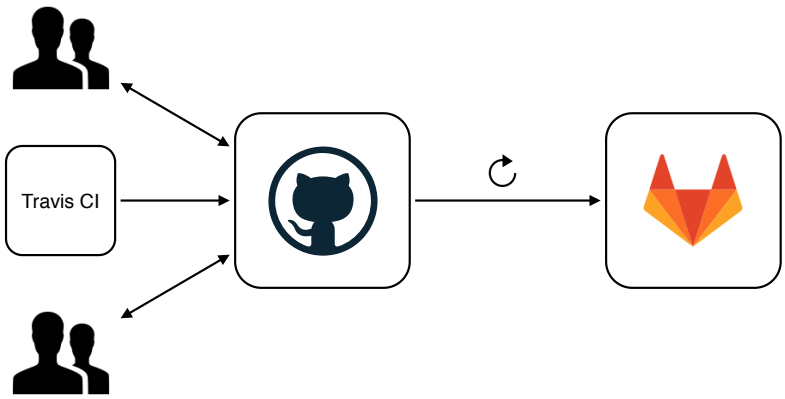

Figure A.2.: We used continuous integration to improve code quality.

As a result, our approach was using “work-in-progress” pull requests, so that there was an open discussion on the progress of certain features. These pull requests would be marked as “passing” if the branch was buildable. We could have extended this by running automated tests, however we did not find a feasible way to customize the environment well enough to reflect detailed hardware parameters of our system.
On a regular basis, the GitHub repository was mirrored by an external server into the GitLab repository provided by the lecture. We closely monitored our custom setup so

that all team members could rely on having the code synchronized with the official code sharing platform.

B. User Guide

To get started, we need to setup the hardware, build the operating system, and flash it onto the microcontroller. After that, a shell provides a user-friendly interface for launching programs and navigating the file system.

B.1. Hardware Setup

First, we need to connect our board with our personal computer using two micro-USB cables. One needs to be plugged into the X4 port, and is used for USB UART and power. The second cable needs to be connected to the X10 port. It is used to download image into memory.
Then, a button needs to be connected to JP1. In our experience, the board will not get powered without this connection.
The device will appear as `/dev/ttyUSB0` on the system. When using a user without root permissions, we need to setup udev rules use the device due to lacking permissions.

B.2. Building and Installing

Our team made use of the provided Docker image for building the operating system.
There are two Makefiles provided with our repository, `Makefile.docker` and `Makefile.podman`, for using Docker or Podman respectively. To use either, create a symbolic link. The majority of our team used Podman throughout the course. The advantage is, that no privileged permissions are required.
Next, type `make hake` to generate the Makefile, and `make build` to build the image. Finally, reset the board and issue a `make install`.
If PyUSB is not installed on your system, you will get an error. Instrument the provided `Pipfile` to install the required dependencies.
To see the serial output, you can type `make monitor` to start minicom in an appropriate configuration.

B.3. Shell

You find the source tree of aosh in `/usr/aosh/`. Build from source and you'll find the compiled binary in `/armv8/sbin/aosh` in your build directory. AOSH relies on the nameservice and monitor to be available and uses the serialservice to access I/O. Make sure these services are running and spawn `aosh`[1].

[1] This is already done in the submitted code.

B.3.1. Built-In Functionality

AOSH comes with built-ins. Issue 'help', which is a builtin itself, and you'll get all built-ins available. Listing 13 shows all implemented built-ins. Their functionality is mostly self explanatory and inspired by UNIX. Where needed, a help flag (`-h`) is implemented to support further usage instructions. See section 9.3 for more implementation details and design decisions.

```
aosh >>> help

list of commands:
> help...........: prints this help
> clear..........: clear screen
> echo...........: display a line of text
> oncore.........: spawn a dispatcher on a given core
> time...........: time a command
> ps.............: report a snapshot of spawned processes
> pid............: show my pid
> coreid.........: show my coreid
> rpctest........: testsuite for rpc-tests
> run_memtest....: runs memory write/read test
> color..........: color test in terminal
> nslookup.......: lookup a service at the nameserver
> nslist.........: list services registered at the nameserver
> ls.............: list directory contents
> cat............: concatenate files and print on the standard output
> cd.............: change directory
> pwd............: print current working directory
> rm.............: remove file
> mkdir..........: make directory
> rmdir..........: remove directory
> touch..........: create file
> ip.............: show my ip
> exit...........: exit shell (ctrl-d)
```

Listing 13: Issue help for a list of supported built-ins

B.3.2. Spawning Domains

In order to spawn domains, the built-in `oncore` is provided. Oncore spawns a new domain on a given core. An optional `coreId` can be passed. Flag `-f` suspends the shell until newly domain finishes. `-t times` spawns a domain `t` times.

```
aosh >>> oncore -h
oncore runs a dispatcher on a given core
usage: oncore [-c coreId] [-t times] [-f]  name
flags:
```

```
-f: run dispatcher in foreground and block shell
-c: set core id
-t: number of domains to spawn
```

B.3.3. Running Serial I/O Tests

There is a binary in `/armv8/sbin/serial-test` in your build directory. This demonstrates how serial I/O is de-multiplexed among domains. The binary spawns itself multiple times and its children access the serial RPCs to all read from I/O. Sessions are acquired by a new domain and released if a new-line character is hit. The tests spawns itself on both cores.

```
aosh >>> oncore -f serial-test
spawning 'serial-test' on core '0'
Shell is waiting until pid 106 has exit
[pid: 106] You may want to run this with oncore -f flag such that shell is
↪  suspended
[pid: 106] This test demonstrates how serial getchar is demultiplexed among
↪  domains
[pid: 106] spawning children...
[pid: 106] Spawning child with pid 107
[pid: 106] Spawning child with pid 108
[pid: 107] Write something and hit return
[pid: 108] Write something and hit return
[pid: 106] Spawning child with pid 109
[pid: 106] Spawning child with pid 110
[pid: 106] Waiting until children have exited
[pid: 109] Write something and hit return
[pid: 110] Write something and hit return
[pid: 107]  o
[pid: 107]  n
[pid: 107]  l
[pid: 107]  y
[pid: 107]
[pid: 107]  p
[pid: 107]  i
[pid: 107]  d
[pid: 107]
[pid: 107]  1
[pid: 107]  0
[pid: 107]  7
[pid: 107] typed: 'only pid 107'. exiting now...
[pid: 108]  n
[pid: 108]  o
[pid: 108]  w
[pid: 108] typed: 'now'. exiting now...
...
```

B.4. Nameserver

To show all registered services the `nslist` command can be used. As an example, here are the registered services on a newly booted instance.

```
aosh >>> nslist
There are 5 services matching query '':
serverinit
servermonitor0
servermonitor1
serverprocess
serverserial
```

To check the PID of a process that provides a certain service the `nslookup` command can be used.

```
aosh >>> nslookup servermonitor0
Service provided by process with PID 0
aosh >>> nslookup servermonitor1
Service provided by process with PID 1
```

With `ps` we can check which processes these PIDs correspond to.

```
aosh >>> ps
   PID    STATUS        NAME
     0    active        init0
     1    active        init1
    10    active        processserver
   100    active        initserver
   101    active        serialserver
   102    active        enet
   103      exit        blockdriverserver
   104      exit        filesystemserver
   105    active        aosh
```

With that we can see that the service `servermonitor0` is provided by the process `init0`, which is the init process of core 0 and the service `servermonitor1` is provided by the process `init1`, which is the init process of core 1.

Running the Test Process

The provided `nameservicetest` has been adapted to register more than one service and also test the deregistration of a service. It can be run by issuing `oncore nameservicetest`. The test will first register multiple services, deregister one of the services and register it again. It will also send one message to each of the registered services and receive a reply for each message.
Afterwards one can check the registered services again with `nslist`, which now includes the additional services spawned by the test.

```
There are 11 services matching query '':
serverinit
test/bla12
test/bla31
test/bla32
test
test/bla1
servermonitor0
servermonitor1
serverprocess
test/bla
serverserial
```

At this point it would also be good to filter this list of registered services. This can be done by adding a query to the `nslist` command. For example, to list all services starting with `test`

```
aosh >>> nslist test
There are 6 services matching query 'test':
test/bla12
test/bla31
test/bla32
test
test/bla1
test/bla
```

B.5. Filesystem

The filesystem can be used using the shell built-ins (see above) or by dedicated programs that use the libc calls. See chapter 10 for a list of supported libc functions.
To use the file system, the SD card has to be at least 4 GB and formatted with FAT32 and 512 Bytes sectors. This can be done with the following command on a Unix system:

```
sudo mkfs.vfat -I -F 32 -S 512 -s 8 /dev/sda
```

B.6. Networking

To use the networking feature, connect the board with the PC using an Ethernet cable. On the host, a static IP address other than 10.0.0.2 needs to be configured. 10.0.0.2 is already the static address used by the device. During development, 10.0.0.1 was used as the host address.
In the shell, the builtin `ip` will remind the user of the static IP address of the device. Further, the user can spawn the `arp` program, which will print the users IP address and its associated MAC address, and all other address pairs in the ARP cache. This is done by typing

```
aosh >>> oncore -f arp
```

on the shell.
To test UDP functionality, an echoserver using the following command.

```
aosh >>> oncore echoserver
```

Per default, the `echoserver` program will listen for packets on port 9000. Optionally, a port can be specified as parameter. After that, netcat can be used on the host to send UDP datagrams like so.

```
$ nc -u 10.0.0.2 9000
```

C. Hints for Reviewers

Some points that might be helpful to the reviewers of the code

- Read the user guide for functional tests that are executed via the shell
- Consider the note in section 4.7.2 about a possible `paging_alloc` API break
- Nameserver lookups are not supported from the init dispatcher[1] checkout chapter 8
- The filesystem has rudimentary FAT32 support, checkout chapter 10
- In order to use the filesystem wait for the filesystemserver to spawn[2]
- Most of the services are spawned in the `servicelauncher` process.

C.0.1. Grading Callbacks

Here are the locations of the grading function calls:

- `grading_setup_bsp_init` init on core 0 in `usr/init/first_main.c`
- `grading_setup_app_init` init on core 1 in `usr/init/other_main.c`
- `grading_setup_noninit` This is called in the hello dispatcher in `/usr/hello/hello.c`.
- `grading_test_mm` In `main()` function of the first core (`usr/init/first_main.c`) and the `main()` function of the second core (`usr/init/other_main.c`)
- `grading_test_early` In `main()` function of the first core (`usr/init/first_main.c`) and the `main()` function of the second core (`usr/init/other_main.c`)
- `grading_test_late` In `main()` function of the first core (`usr/init/first_main.c`) and the `main()` function of the second core (`usr/init/other_main.c`)
- `grading_rpc_handle_number` Init server (`usr/initserver/initserver.c`)
- `grading_rpc_handler_string` Init server (`usr/initserver/initserver.c`)
- `grading_rpc_handler_serial_getchar` Serial server (`usr/serialserver/serialserver.c`)

[1] Not to be confused with the initserver
[2] The spawn is not blocking because not all members have an SD-Card

- `grading_rpc_handler_serial_putchar` Serial server (`usr/serialserver/serialserver.c`)
- `grading_rpc_handler_ram_cap` Called when allocating memory from the local Memory Manager (`usr/init/mem_alloc.c`)
- `grading_rpc_handler_process_spawn` Process server (`usr/processserver/processserver.c`)
- `grading_rpc_handler_process_get_name` Process server (`usr/processserver/processserver.c`)
- `grading_rpc_handler_process_get_all_pids` Process server (`usr/processserver/processserver.c`)
- `grading_rpc_handler_get_device_cap` Not implemented