O_3 Report

Team 1

Emil Schätzle   Linus Vogel   Shashank Anand   Zikai Liu

June 2^{nd}, 2022

Contents

Chapter 1

Introduction

In this report[1], we present O_3 (Ozone), a multikernel operating system built on part of the Barrelfish codebase. We implement a number of functionalities in Ozone, from low levels to high levels. At low level, we implement the fundamental OS functions, such as physical and virtual memory management and core management. We implement middleware like inter-process communication mechanisms and drivers. And at high level, we implement the shell as well as a handful of other applications. The following list gives an overview of Ozone:

- Physical memory manager
  - The buddy allocation algorithm
  - Binary search tree
- Virtual memory manager
  - The buddy allocation algorithm + free list
  - Support mappings to fixed and dynamic addresses
  - Demand paging
  - Unmapping
  - Thread safe
- Process spawning
- Process management
  - Killing processes
- Intra-core Local Message Passing (LMP)
  - Passing large messages in frame
- User-Level Message Passing (UMP)
  - Efficient ringbuffer
  - Waitset Integration
  - Indirect UMP Capability Transfer
- Remote Procedure Call (RPC)
  - Unifying LMP and UMP channels

[1]Reference: Barrelfish Technical Note LaTeXtemplate and Barrelfish Logo. Additional elements added to the logo are in Common-Creative or public domain licenses [3][4][5]

    - Easy to use interface
- Multicore
    - Bringing up all four cores on iMX.8
- Shell
    - Thread-safe terminal server
    - Command history and Command Line Editing
    - Built-in commands
- Filesystem
    - FAT32 specification with read and write support
    - High speed access
    - Shell integration
- Nameservice
    - Unified interface for RPCs
    - Deregister dead services
- Networking
    - Low-latency on UDP connections
    - Clear user-interface to simple listen on ports and send out UDP messages
    - nchat to chat with another host using UDP

Chapter 2

Memory Manager

2.1 The Buddy System

The memory system should try to satisfy memory requests efficiently while reducing fragmentation. Inspired by the early Linux kernel [1], we employ a modified buddy system as the backbone of the memory system. This system is able to satisfy all requests in poly-logarithmic time (in the size of the address space/number of handed out blocks). At a high level, the buddy system manages naturally aligned memory regions with power-of-two sizes. The minimum size the system can handle is a base page. Requested sizes are rounded up to the nearest power-of-two size. This can lead to an overhead of almost the allocated size (if a little more than a power of two is requested), but yields nice O(1) performance and avoids wasting memory with good alignment. However, only powers of two can be requested as alignments (which is the only practical application). The system then maintains a table of free lists of blocks and tries to find a block of memory that comes closest to the requested size/alignment. If it is too large, the block will be recursively split until it reached the requested size.

2.2 Free List Table

Our implementation uses a two dimensional, triangle-shaped free list table. The possible sizes (powers-of-two) constitute the one dimension, while the other one is formed by the alignment. The triangular shape follows from the natural alignment, as the alignment of a block has to be at least as large as the block size and hence, size/alignment pairs not fulfilling this condition can be omitted (which is one half of the constituted square). The lists are doubly linked, as not always the first block will be removed (see). The rows of the table (which have the same associated block size) are directly adjacent to each other in memory in order to improve locality and increase the speed of a table walk. As we could not find a way of determining the actual size of the address space in user-space (should be 48 bits) we had to use the size of the type representing an address (64 bits), leading to a table size of 1378 entries. As every entry is only a 64 bit pointer, the memory footprint of the table is still small enough (and the last part of it will never be used, as the blocks would be way to large).

2.3 Adding Memory

On startup, init will add memory to the memory manager. This memory will be split into naturally aligned blocks of maximum size which are then added to the according free lists. Each list-entry contains a reference to the added memory capability as well as information on the offset into the associated region and the size/alignment of the block. Note, that the added capability is not split yet (this will only happen when the memory is handed out).

2.4 Handing Out Memory

2.4.1 Finding a large enough Block

Whenever a memory block is requested, a table walk will start at the linked list head, that would perfectly satisfy the request. It then continues first into the direction of better alignments and then in the direction of larger blocks, until a non-empty list is found. This search will access memory consecutively if no special alignment is requested and will only have to skip entries if it is (which should be rather rare). If the search reaches the end of the list, no memory satisfying the request exists anymore. Due to this search pattern, the most "valuable" blocks (i.e. large blocks with good alignment) are only handed out if absolutely necessary which constitutes an huge advantage over simpler approaches such as first-fit.

2.4.2 Splitting the Block

If the found block is too large, it has to be split into smaller blocks. This is done by removing the block from its list, creating two children of equal size (as the size is a power-of-two), adding one of them to the appropriate free list and recursing with the other one. If the alignment of the original block is better than the one requested, the block with the better alignment (i.e., the one to which "fixed address prefix" a 0 was added) will be added to list and the recursion will take place on the other one. Otherwise, the assignment will be the other way around. The runtime of the recursion is bounded by a constant, namely the length of the table. The asymptotic time complexity is $\mathcal{O}((\log n)^2)$, where n is the size of the virtual address space (due to two-dimensionality of the table).

2.4.3 Handing out the Block

After the recursion finishes, we end up with a block of the (rounded-up) requested size but with a possibly better alignment (which will just be wasted). Now, we create a smaller capability out of the original capability which was added to the manager before and return it. Note, that we have to be able to find the block again using the capability that we just created. Therefore, we use a tree which is described in the following section. Hence, we add the block to the tree before returning it.

2.4.4 Pending Tree

In order to find a handed-out (pending) block again using only the capability that was returned, we use a binary search tree. This binary search tree uses the bitwise-reversed capability address of the capability as a search key. The reason to not use the capability address itself is, that it is likely to be increase by a small amount between calls (because that is the order in which capability slots are handed out). This would lead to a tree with (in the worst case) linear depth. However, inverting the order, takes advantage of this behaviour, as (if the addresses were strictly increasing by one between calls) we would end up with almost perfect balancing. Of course, slots might be freed sometimes and not all slots will be handed out the memory manager, but overall the balancing should (and does so in practice) turn out quite nice, leading to a low two-digit depth. In order to avoid destroying the balancing when deleting nodes (which we will have to do), we also employ a pseudo-random heuristic (currently parity of the key) to choose whether to move the predecessor or successor up to replace the node.

Assuming that these heuristics work somehow, the time of this operation can be assumed to be in $\mathcal{O}(\log n)$ (where n is the number of pending blocks). However, the worst-case runtime is still $\mathcal{O}(n)$ (i.e., the binary tree becomes a linked list). One could get stronger runtime guarantees, by using red-black trees (for instance), but during all of our experiments we never found the memory server performance to be slow (in fact it was fast). Also, we would have to implement the red-black tree ourselves due to the peculiarities of our implementation (see 2.6) which would be very time consuming. Lastly, better asymptotic bounds do not imply better practical performance.

2.5 Taking Back Memory

When memory is handed back to the memory server, a find operation on the pending tree is performed and the capability is revoked. Then, as long as we have a parent (i.e., a node our node resulted from during a split) and our ”buddy” (the block that was created during the same split as our block) is in a free list, the two blocks will be merged together again and we recurse on the merged block. After this recursion, the final block will be added to the corresponding free list again. For this to work, some additional pointers (and booleans encoding states) are required for every element in the tree.

This operation is clearly linear in the maximum number of splits that could have been performed and hence, logarithmic in the size of the address space.

2.6 Optimizing the Datastructure

We implemented a few optimizations to make the datastructure fast: Firstly, as mentioned before, we arrange the rows of the table to be consecutive in memory and allow for fast scanning. Secondly, we implement some functions using gcc intrinsics (such as determining the alignment of an address, inverting the bit-order or calculating the parity). Lastly, we use a single datastructure (node) to be used in the list and the tree. This way, we are able to reuse the pointers without reallocating nodes all the time and were able to fit every node into a single cache-line (64 bytes).

2.7 Avoid Running out of Memory

As we run in the memory manager itself, it is not possible to simply malloc nodes if we need them. Hence, we have to make sure, that we always have enough slabs (which are used for nodes) left to split a very large block into a very small one (and also enough slots to make new slabs out of the small block even if we need some for new L-Nodes). The number of slabs required can be bound by twice the maximum depth of the recursion + some constant safety margin. For the slots preserving a small constant amount is sufficient.

Note, that taking care of running out of slabs sometimes means to have a nested node allocation. We had to keep this in mind when ordering instructions. For instance, we have to remove a node from the free list before we can allocate slabs for its children, because allocating them might trigger a slab refill which would then find (and use) the same node, without us noticing after the allocation returned.

2.8 Some History: Our First Attempt

At the very beginning of the group projects, we decided on one of our memory managers to be the basis of our common system. The implementation we chose then was heavily based on a the malloc model of memory management and handled physical memory much the same way malloc does with virtual memory. It kept track of allocated and free blocks in a linked list of blocks and was capable of splitting and coalescing them as needed. Over time however, we noticed not only some serious bugs, because the complex linked list structure was hilariously hard to get correct, but we also noticed some serious drawbacks and issues including for example memory fragmentation, and most noticeably serious performance issues with only small amounts of memory ever handed out. Since we wanted to get the memory manager right, as it literally is the basis of our system, we opted to completely replace this initial implementation, that was still heavily dependent on our first ever attempt, before the group projects even, with a new implementation that draws from the better understanding we have gained over the system.

Chapter 3

Virtual Memory

The virtual memory system (paging system) manages page tables and memory regions in the virtual address space. It supports fixed mapping, dynamic mapping, unmapping, and self-paging (demand paging). As the cornerstone of every application running on Ozone, the paging system needs to be reliable and efficient. We employ multiple data structures and algorithms, including doubly-linked lists, red-black trees, and the buddy system, to achieve efficient mapping and unmapping operations. In this section, we will discuss the system in detail.

3.1 VNode Management Subsystem

Ozone, with its foundation on Barrelfish, uses the capability system to manage page tables [6]. To perform memory operations, we need the corresponding VNode capabilities.

ARMv8 features 4-level page tables and therefore there are 4 levels of VNodes. We manage them with 4 red-black trees (RB tree), using the virtual address as the index. RB tree data structures and functions are instantiated from the FreeBSD tree library [2], which is distributed as part of the Barrelfish codebase. Level 0 RB tree contains a single element of the L0 VNode.

RB tree is a balanced binary search tree, providing $O(\log n)$ complexity for inserting, deleting, and searching in the worse case. Since VNodes are indexed by their virtual addresses and the address space is limited to 48 bits, the complexity is basically bounded by a constant.

VNodes are constructed on demand. For example, when the system tries to map a basic page, it looks up the corresponding L3 VNodes in the level-3 RB tree, whose address can be trivially calculated by masking out the low bits. If the L3 VNode is not found, the corresponding L2 VNode is looked up or constructed as needed, and is used to construct the L3 VNode. The L0 VNode is retrieved from the well-known location from the CSpace when initializing the paging system.

At the early stage of our development, we have considered managing VNodes using multiple levels of arrays. Each VNode corresponds to an array of 512 child VNodes, which can be lookup in $O(1)$. However, there are two major challenges with this approach. The first one is the waste of space. Each `capref` takes 16 bytes so every array takes $512*16B = 8192B = 8KB = 2$, even if only one entry is used. Another challenge is bootstrapping. To perform the very first mapping, arrays of L0 to L3 VNodes need to be constructed, which already takes $32KB$. All such space needs to be statically allocated since the paging system is not working yet. Slab allocators are not designed to allocate objects larger than a page, while working with static buffers introduces a different control path. Later, We turn to RB tree as a more efficient and robust solution.

3.2 The Buddy System and Free Lists

The virtual memory system should try to satisfy memory requests efficiently while reducing fragmentation. Inspired by the early Linux kernel [1], we employ the buddy system as the backbone of the virtual memory system.

At a high level, the buddy system manages naturally aligned memory regions with power-of-two sizes. The minimum size the system can handle is a base page. Requests are rounded up to the nearest power-of-two size.

When allocating a naturally aligned memory region of size 2^x bytes, the system looks for a region of the exact size in the level-x free list. If no such a free region is available, the system continues looking for 2^{x+1}, 2^{x+2}, and so on, until a free region is found. The system then takes the first 2^x bytes of the region and systematically breaks down the remaining to power-of-two regions. For example, Figure 3.1 shows an example of allocating 64 pages from a 256-page region. The number of splits is $O(\log \frac{n}{m})$ where n is the size of the region to be split and m is the requested region size. As the maximum region is 2^{48} bytes (the whole virtual memory space) and the minimum is 2^{12} (4KB, a base page), the complexity is bounded by a constant.

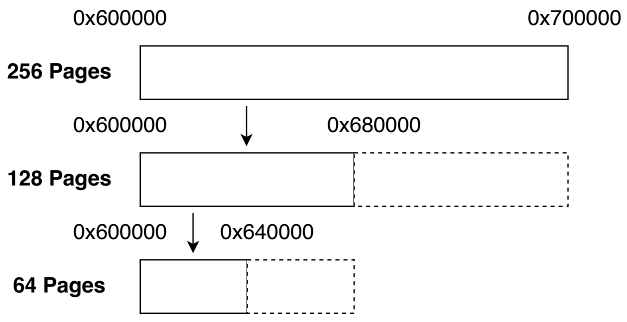

Figure 3.1: Example of splitting a 64-page region out of a 256-page region.

When a memory region is split by half, each of the resulting two regions becomes `buddy` to the other. Given a region of size 2^x and start address v, the address of its buddy can be easily calculated as

$$v \text{ XOR } (1 << x).$$

For example, in Figure 3.1, the buddy of the region $0\text{x}680000$ is $0\text{x}680000 \text{ XOR } (1 << (7 + 12)) = 0\text{x}600000$, and vice versa.

When a region of size 2^x is deallocated, the system checks if its buddy is also free. If so, two regions are combined and become a naturally aligned region of size 2^{x+1} bytes. This process is repeated until the region can no longer be combined. Similarly, the number of combinations that can happen in single deallocation is bounded by a constant.

By systematically breaking down memory regions, the buddy system guarantees that the fragmentation rate never exceeds 50% [1]. In practice, the most common memory requests are one or a few base pages, and therefore there will be much less fragmentation.

In our implementation, each memory region is represented as a node (`struct paging_region_node`) with a start address and a size (in bits). Figure 3.2 shows the data structure. Region nodes are created on demand and stored in an RB tree indexed by the start addresses. This means that regions with the same start address but different sizes, for example, $[0, 0x1000)$ and $[0, 0x2000)$, are represented by the same node with address 0 depending on whether the size is 12 or 13 bits. This is an intentional design choice since these regions cannot be free at the same time. If a region node of size 2^x exists, it implies that region $2^{x+1}, 2^{x+2}, \ldots$ are already split.

To efficiently find available regions of specific sizes, in addition to being in the RB tree, free regions nodes are chained in doubly-linked lists called `free lists`, as shown in Figure 3.2. Since the maximum region is 2^{48} bytes and the minimum is 2^{12} (4KB, a base page), there are $48 - 12 = 36$ free lists. When

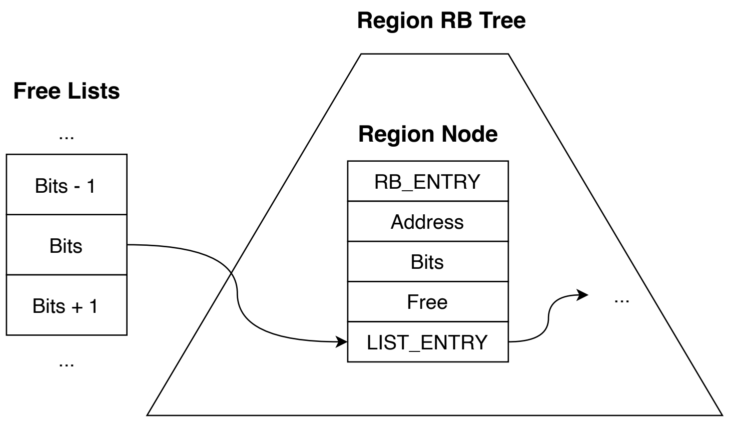

Figure 3.2: Data structures of memory region management.

looking for a region of a specific size, the system can retrieve the list head if the list is not empty in $O(1)$ time. The system accesses at most 36 free lists to find a free region that is large enough to accommodate the memory request, or gives up if there is no enough memory.

3.3 Fixed Mapping

The paging system supports mapping a frame to a user-specified address, which we call fixed mapping. In the scenario of fixed mapping, the system should map the frame exactly, without any rounding up. However, the frame size is not necessarily a power of two and the region may not be naturally aligned. In this case, the frame needs to be systematically broken down before the buddy algorithm applies. Figure 3.3 shows an example of mapping a frame of size $352 = 16 + 64 + 256 + 16$ pages to address `0x5B0000`. As long as the frame size is divisible by the base page size (guaranteed for Frame, while DevFrames are rounded up to the nearest multiple), the frame is mapped exactly. The actual mapping operations are then performed on these naturally aligned memory regions, using the buddy algorithm discussed above.

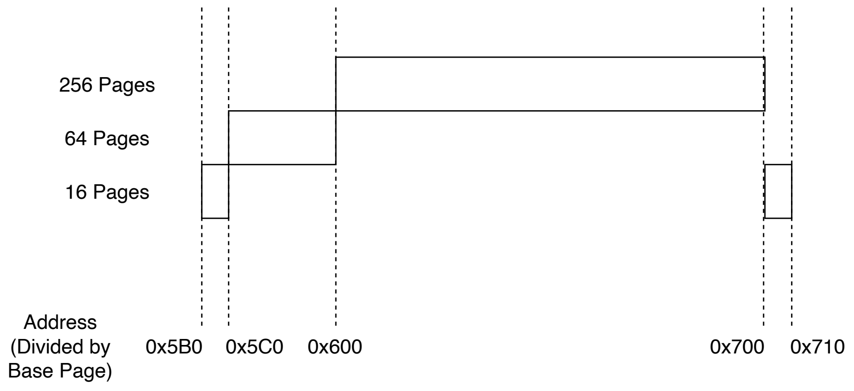

Figure 3.3: Example of systematically breaking down the region $[$`0x5B0000`$,$`0x710000`$)$ to natually aligned power-of-two regions. Addresses in the image is divided by the base page size for clarity.

3.4 Dynamic Mapping

With dynamic mapping, the paging system is free to choose the virtual address. Our paging system allows the user to specify the required alignment of the address, but the alignment must be power-of-two and a multiple of the base page size. Assume the system is requested to map a 2^x-byte frame with alignment 2^y. If $y \leq x$, the system looks for a region of size 2^x and larger regions if needed, using the buddy algorithm discussed above.

If $y > x$, that is, the alignment is stricter, the system has a fast path and a slow path. For the fast path, it starts directly from 2^y-byte regions. For every region larger or equal to 2^y bytes, the alignment is automatically satisfied, since all regions are naturally aligned. Therefore, the system only needs to check the head of each free list. If the fast path fails, the system falls back to the slow path, where it starts again from 2^x-byte regions and goes into the linked list to see if there is a region that satisfies the requested alignment. It is possible that a region smaller than 2^y bytes (but larger than 2^x bytes) has an alignment of 2^y, since its buddy is not free for example. But in this case, the number of lookups is unbounded.

3.5 Unmapping

To support unmapping, the paging system needs to store the Mapping capabilities associated with each allocated memory region. There can be multiple Mapping capabilities for one region, when the region spans multiple L3 page tables or is constructed with self-paging (discussed in the next section), for example. To efficiently support unmapping, we add another RB tree to manage the Mapping capabilities.

Each mapped region has an associated `mapping node` (`struct paging_mapping_node`). Each region node holds a pointer back to the region node and a doubly-linked list of `mapping child node` (`struct paging_mapping_child_node`), where the Mapping capability, the associated VNode of the Mapping, and the Frame capability if the region is constructed by demand paging (discussed in the next section) are stored. Mapping nodes are managed by the RB tree using the virtual address of the region as indices. The lookup complexity is again $O(\log n)$ and bounded by a constant due to limited virtual memory space.

To unmap a memory region, the corresponding mapping node is looked up using the start address of the region. The Mapping capabilities are unmapped and destroyed by going through the list, which takes $O(m)$ times where m is the number of the Mapping capabilities. For a fixed or dynamic mapping, m equals the number of L3 page tables that the region spans, as the paging system applies mapping in one L3 table all at once whenever possible. For a demand paging region, m equals the number of base pages that are actually mapped.

3.6 Demand Paging

Our paging system supports allocating large regions that only install physical pages on demand in response to faulting accesses into the region, which is known as lazy allocation or demand paging.

Given the existing framework of our paging system, demand paging is easy to implement. When allocating a region, the system finds a suitable region from the free list, possibly by splitting a larger region, and claims it as used, without performing any frame mapping or page table construction. We call the region a `placeholder`. Finding and splitting regions take a constant bounded time as discussed above. A corresponding mapping node is created.

When a page fault occurs, the exception handler first checks the address against NULL, a common real page fault. If the address is not NULL, the handler looks up the mapping node by gradually masking out low bits of the address. In other words, it finds the placeholder region where the address is in by doubling the region size every time. It takes at most 36 trials before the address is completely masked out, while each RB tree lookup takes $O(\log n)$ time. Once the region is found, the mapping takes $O(1)$ time. The Mapping capability and The frame capability are stored in a newly constructed mapping child node, which is then added to the head of the mapping node in $O(1)$ time.

If the address does not fall in a placeholder, the handler reports a real page fault. By using the RB tree to look up mapped regions, the system is able to strictly check the faulting address (i.e. not installing a frame to somewhere not mapped) while maintaining high efficiency.

In fixed and dynamic mappings, the paging system does not manage the life cycle of the frame being mapped since it is supplied by the user. For demand paging, installed frames are invisible to the user. Therefore, the system stores the frames in the mapping child node and destroyed them when the region is unmapped.

3.7 Thread Safety

Data structures in the paging system are protected by respective mutexes. Namely,

- VNode RB tree is protected by a mutex region when looking up or creating VNode,
- region node RB tree and free lists are protected by a mutex when looking up, splitting, or combining regions, and
- mapping node RB tree is protected by a mutex when performing mapping or unmapping.

3.8 Limitations and Known Issues

3.8.1 Dynamic mapping always allocates the size of the nearest power of two

In our current implementation, dynamic mapping (including demand paging) always rounds up the requested size to the nearest power of two. In theory, the requested region can be systematically broken down in the same way as fixed mapping and the tailing free space can be reused. We haven't looked into it due to the time limit.

3.8.2 Thread safety when mixing different types of mappings are not yet thoroughly tested

Data structures in the paging system are protected by mutexes. While we have stress-tested multi-threaded demand paging, we have not thoroughly tested mixing different types of mapping (fixed, dynamic, demand) with multiple threads.

Chapter 4

Process

In this chapter we describe the work done to spawn a process and manage spawned processes in Ozone.

4.1 Paging Extension

Refer to Chapter 3 Virtual Memory where the details of the paging system are described in full detail.

4.2 Process Spawn

To spawn a process, we must perform the following steps

1. **Setup CSpace**: Create all the named capabilities in the well known locations
2. **Setup VSpace**: Create the root page table vnode and the corresponding cnode
3. **Find and map the ELF binary**: The cap is obtained from cnode_module. Map the ELF to our address space
4. **Parse the ELF**: Load the elf and get the address of the .got section
5. **Setup Dispatcher**: Create the dispatcher cap and fill in information about the process
6. **Setup LMP Endpoint**: Create an LMP channel to init and place it in a predefined location in the taskcn of the child process
7. **Setup Arguments**: Parse the arguments, fill a capability with them, and place it in a well known location in the child's cspace
8. **Start Dispatcher** Invoke dispatcher with the created dispatcher cap

At the end of all these steps, the kernel takes over and completes spawning the process.

4.3 Process Management

In Ozone, process management is integrated with the spawn library. One major motivation is that the spawning API returns domain ID (or process ID, PID) upon a successful spawning. If we use a single incremental counter to assign PIDs, we will ultimately run out of PIDs in the long term. In contrast, by integrating the process management system, the spawn library can reuse the PIDs of terminated processes.

The process management system in Ozone is distributed. Every init process holds a list of processes running on the current core. The main motivation is that init on each core needs to hold the LMP channels to all the processes, in order to transfer capabilities across cores (discussed in the UMP section).

Introducing a centralized copy of all the processes does not eliminate the need for the local copies, but adds the overhead of synchronization. Therefore, we keep the process lists distributed. To get the list of all processes running in Ozone, init queries the other cores for their local process list, aggregates them, and returns them to the user.

PIDs are unique across the OS. For both readability and easy message forwarding (discussed in the UMP section), PIDs are constructed using the following rule:

$$\text{core ID} * 10000000 + x,$$

where x is a number between 1 and 10000000 (exclusive) that is unique among the living processes on the same core. $x = 0$ is reserved. For example, processes on core 1 are numbered as $10000001, 10000002, \ldots$. Given a PID, the core ID can be easily calculated as PID$//10000000$.

On each core, each living process is associated with a node in a red-black tree implemented with FreeBSD's tree library [2]. Insertion, deletion, and looking up all take $O(\log n)$ in the worse case, where n is the number of living processes on the core. Each process node holds the process name, the PID, the dispatcher capability, and the LMP channel with init.

4.4 Stop a Process

To kill a running process, we introduce a new invocation on the dispatcher capability: Dispatcher Stop. In the kernel, this invocation causes the dispatcher to be removed from the scheduler. Init then destroys the dispatcher capability and removes the process from the process manager. The PID is reused by inserting it into a free list of PIDs.

4.5 Limitations and Known Issues

4.5.1 spawninfo is not cleaned up after the child process starts

In the current implementation, the resulting spawninfo is not freed after a child process starts. The main reason is that unmapping was not yet supported in our paging system by the time we developed the spawn library. To perform the cleaning, slots taken by the capabilities of the child process should be freed, and the binary mapped in the parent's virtual address space should be unmapped.

4.5.2 Cleaning up zombie processes

When a process terminates, it notifies the init process by sending an RPC message (discussed in the RPC section). However, the process may not be completed clean up. In the provided codebase, we have seen code scanning the process's CSpace and destroying the capabilities. However, we have not yet registered the monitor endpoint to the kernel so the memory capabilities may not be correctly recycled.

Chapter 5

Local Message Passing (LMP)

The Barrelfish codebase provides the interface of LMP endpoints and channels for intra-core communication. Messages go through the kernel and are placed into the endpoint of the receiver process. The provided interface allows sending messages with a maximum of four 64-bit unsigned integers as the payload, plus an optional capability. For ease of use by application, Ozone extends the LMP interface by creating wrappers for the sending and receiving APIs. We discuss these extensions in this section.

5.1 LMP Message Format for RPC

To make full use of the available payload space, we decide to treat the payload in byte granularity, namely 32 bytes, rather than four integers. Specifically for remote procedure calls (RPCs), typically we want the exact size of the payload in bytes and an identifier to dispatch messages passed on the same channel. Accordingly, we take the first byte to store the size (in bytes) of the payload, and the second byte as the `identifier`. The remaining 30 bytes are used to pass the user-specified payload. One byte is sufficient to store the size (up to 30). The highest bit of the identifier is reserved for indirect UMP capability transfer, and it can take one of the 128 different values. The format is visualized on the left of Figure 5.1.

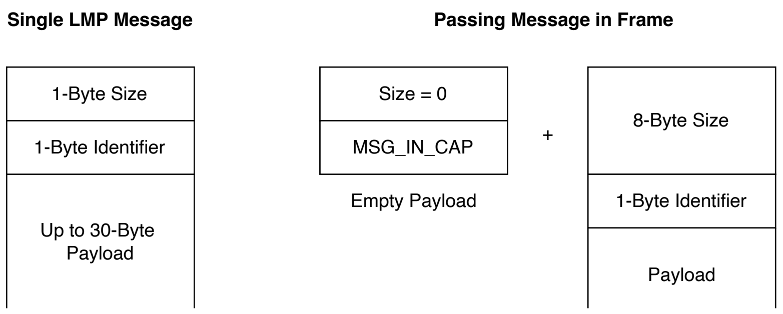

Figure 5.1: Formats of LMP messages designed for general RPC.

5.2 Passing Large Messages Through Frames

The maximum payload that can be passed in a single LMP message is 30 bytes, using the format mentioned in Section 5.1. To pass a large message, one approach is to split the payload into multiple messages and reassemble the payload on the receiver side. However, this approach can be inefficient if the message is very large since every transfer can involve two context switches between the kernel and the user programs.

We implement another mechanism that transfers large messages through frames. When the payload is larger than 30 bytes, the sender acquires a frame, copies the payload, and sends the frame as the capability along with an empty LMP payload. The payload size is theoretically unlimited since the frame can be arbitrarily large given the available memory. Figure 5.1 shows the format used in this case on the right side. At the start of the frame, the size is encoded in a 64-bit unsigned integer, since the payload can be larger than 255 bytes now. The identifier of the LMP message is replaced by a special one indicating the payload is passed in the capability. Combined with the receiver wrapper discussed in the next section, this process is transparent to the user.

Passing large messages in frames is only more efficient than sending multiple messages when the payload is large enough since allocating a frame also involves an RPC call to init. In practice, most usages of LMP only work with messages less than 30 bytes, while UMP is preferred when high-performance bulk transfer is needed.

5.3 LMP Message Handler Wrapper

When working with LMP channels and the format discussed in Section 5.1, message handlers do much work in common. The handler needs to read the actual payload if it is passed in a frame, deserialize the size and the identifier, refill the capability slot if it is consumed, and specifically for RPC, reply to the call with another payload or the error code, and perform the cleanup. We put all the work into a unified handler for LMP messages in `libaos`, which call back the user-specified RPC handler. We will further discuss the RPC procedure in the RPC section.

5.4 Limitations and Known Issues

5.4.1 Cannot pass capability if a large message is passed in a frame

In our current implementation, the user is not allowed to transfer an additional capability when the message is passed in a frame, since the capability slot is taken by the frame capability. If the message is too large and a capability is supplied, the function returns an error. Without changing the existing code, the user can choose to send the message and the capability respectively by making two calls. Alternatively, the library can be adjusted to send and receive a subsequent message taking the capability following the frame.

Chapter 6

User-Level Message Passing (UMP)

6.1 The Ringbuffer

A single direction of a URPC channel consists fundamentally of three components:

1. A data sink that carries data from the sender
2. A data source that carries data to the receiver
3. A medium that transports the data in between the above two

In this section, we want to focus on the 3rd of the above, which our system implements as a ringbuffer. Not only is this the structure that the book has described, but also a ringbuffer offers little overhead, both in memory and compute contexts, and is even a simple to implement concept. And thus, we had a ringbuffer implementation intended to be used in a shared frame with a single data produces sitting in the sender process, filling the ringbuffer with data, and a single consumer, emptying the ringbuffer again. Ideally this approach would be duplicated and copied to implement the reverse direction of the channel.

However, the next few paragraphs of the book highlight a slight problem with this approach: The two ends are not in the same process, and not even necessarily run on the same core. This brings the issue of memory consistency into play here and honestly, this is a bit hard to fully grasp.

The book outlines how a data transfer through such a shared ringbuffer could look like, and even claims that the example is how it should be done on ARMv7. Following this, there would need to be a flag next to the data that would get set by the producer only when the data has been completely written, and the data would only be read, once the flag has been set. This procedure would rely on the cache coherency protocol that would invalidate cachelines present on other cores once written to. This means that the programmer would need to add so called memory barriers in between the main read/write operations and the clearing/setting of the flag, such that the order of memory operations is poreserved, i.e. the order 'write data', 'write flag', 'read flag', 'read data' would be preserved. This is necessary, because the ARM architecture does not in general guarantee that memory accesses happen in the same order that the program could would lead you to believe.

In addition, the book mentions that the ARMv8 memory consistency model has made such situations more intuitive to handle for the programmer, however, after extensive reading in the ARMv8 reference manual, we could not find any difference that would arise in this case where a flag is used for synchronisation, and thus we kept the same approach as the book outlined.

Figure 6.1 depicts how our implementation works on a high level. The ringbuffer has a size of 4096 bytes and its entires, i.e. the units that are written and read, are 64 bytes in size. We chose 64 bytes, since this is the cacheline size of the processor and the the unit that is synchronized via the cache coherency protocol. The additional two barriers before writing the data to the buffer and after reading data from the buffer, are in place because the buffer may get completely filled, in which case the dependencies

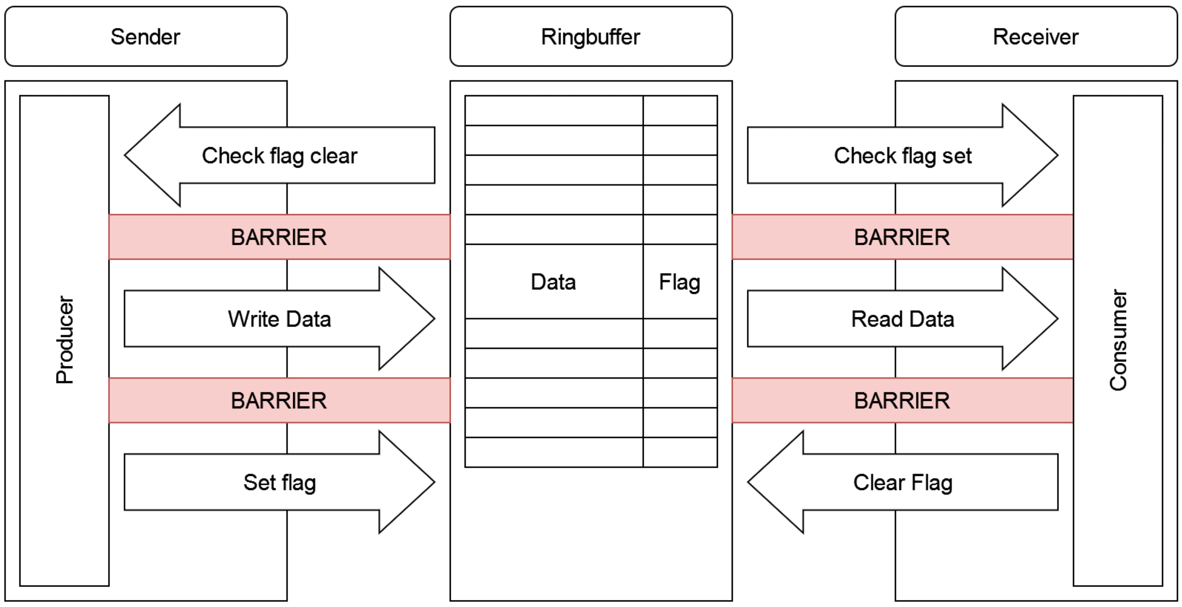

Figure 6.1: Moving data through the ringbuffer

reverse and the producer has to wait until the consumer has read the next chunk of data. Also, since we want the rungbuffer to fit into a single page, there are only 63 entries in the buffer because the space of the last entry is used for meta information about the buffer, such as its current head and tail, that is needed to for the buffer to function.

Even though our implementation of this ringbuffer does work correctly, ultimately it is not quite enough just to be able to pass chunks of 56 bytes of data from one end to the other. Since this ringbuffer should be the foundation of interprocess communication, it has to be slightly more sophisticated than that. In order to properly send arbitrary data reliably from one process to another we also need the means of splitting the data into smaller chunks and correctly reassembling it at the other end. To this end, we created a layer between any clients of the ringbuffer library and the actual ringbuffer, consisting of the `ringbuffer_producer` and `ringbuffer_consumer` interfaces, and hid the buffer itself behind them. This producer/consumer infrastructure is what the programmer may use to handle a given ringbuffer, while most of the direct interface of the ringbuffer itself is not visible to the client.

Data fragmentation and reassembly is completely handled within the producer/consumer constructs and completely transparent to the user. Any message passed to the ringbuffer producer will be prepended its length and then split up into 56 byte chunks, which will then be passed into the ringbuffer. On the other end of the channel, the ringbuffer consumer reads chunks out of the buffer as they arrive and combines them in a buffer that can be dynamically allocated to fit the received message, since its length has been prepended to the message. Once the transfer is complete, this buffer is then handed to the client who is responsible for freeing the buffer again, when it is no longer needed.

This transfer is very general and really capable of passing arbitrary amounts of data from producer to consumer. Even though the ringbuffer itself is only a single page in size (keep in mind that this is onle one direction of the channel, the reverse direction of the channel uses another instance of this ringbuffer), even messages much larger than one page can be passed through it. However, in such situations it is important to know that both sides of the transfer will essentially block until the entire transfer is complete (assuming that a 4kB is negligible compared to the message size, of course the producer may return before the consumer has consumed, the last 4kB of data).

6.2 Arbitrary UMP Channels

We build bidirectional UMP channels on top of the ringbuffer. Each UMP channel is two ringbuffers next to each other, taking two base pages shared by two processes. On one side, the `ring_producer` is instantiated on the first page and the `ring_consumer` on the second page, while on the other side they are reversed, as shown in Figure 6.2.

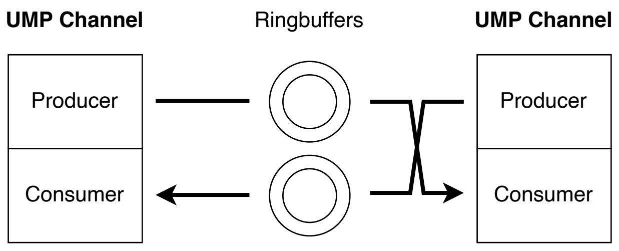

Figure 6.2: Build a bidirectional UMP channel on two ringbuffers.

To set up a UMP channel, the shared frame needs to be received by both sides. For inter-core communication, the frame capability of one side needs to be forged by the monitor. Since capabilities cannot be directly transferred through the UMP channel, we design the UMP binding process to be third-party coordinated. In other words, a `coordinator` process allocates the shared frame capability, passes it to both processes by any mean, and each process instantiates the UMP channel on the frame. The coordinator can be one of the two processes, or init, or any other process. The frame can be passed at a well-known location, through LMP, or by serializing on one init and forging on the other.

Since both sides are unaware of whether the other side already has set up the channel and possibly has written messages into it, neither side can memset the shared frame to zero, otherwise, messages can get lost. Therefore, the coordinator is responsible for zeroing the frame before passing it out. In practice, frames created by `frame_alloc` are automatically zeroed by the kernel, so typically no extra work is needed.

6.3 Waitset Integration

We integrate UMP channels as polled channels of waitsets so that they can be dispatched the same way as LMP channels. The codebase already supports polled channels. We add `struct waitset_chanstate` to the UMP channel structure and check if there are messages available in the ringbuffer in `poll_channels_disabled` .

6.4 UMP Message Format for RPC

UMP channels are also used for remote procedure calls. To allow unified handling of RPC messages, the one-byte identifier used in the LMP message format (Section 5.1) is also prefixed before the actual payload in UMP. The size is not prefixed since the ringbuffer handles the message size internally. Similar to the LMP system, we also implement a generic UMP message handler that performs the common operations and calls back the user-specified RPC handler. We will further discuss the RPC procedure in the RPC section.

6.5 UMP Capability Transfer

Capability cannot be directly transferred through UMP channel. And inter-core capability transfer can only be performed by the init processes since only they hold the monitor capability to forge capabilities.

Inspired by Barrelfish, we develop a similar init-assisted UMP capability transfer mechanism. Figure 6.3.

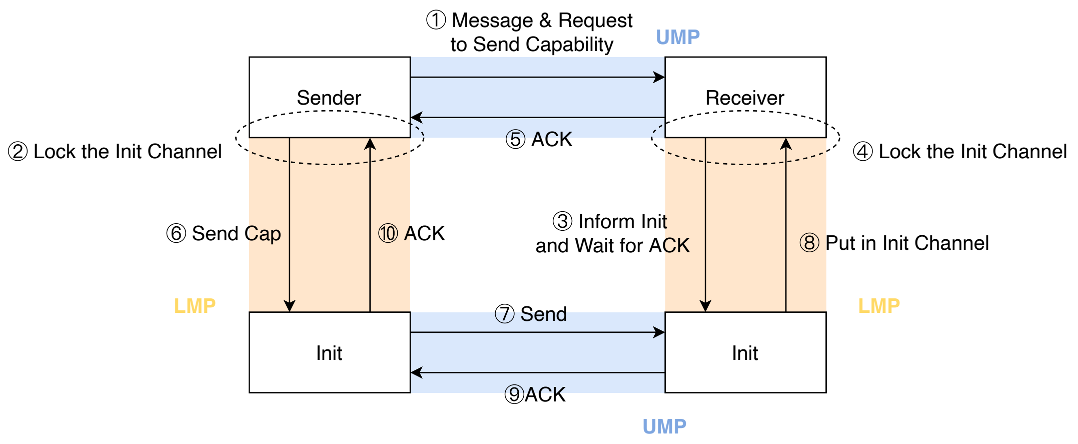

Figure 6.3: Indirect capability transfer in UMP channel through init.

A. The message payload is passed directly through the UMP channel. The highest bit of identifier is set to indicate that a capability is going be sent.

B. The sender grabs the lock of the LMP channel to init.

C. The receiver notifies the init process that there is an incoming capability. This step is essential to avoid a buggy or malicious process to send arbitrary capability to the process and consume the endpoint buffer and the capability slot without the contest from the receiver.

D. The receiver grabs the lock of the LMP channel to init after making the RPC call.

E. The receiver sends an ACK on the UMP channel.

F. The sender sends the cap to init with the receiver's PID with an init RPC call.

G. The init process checks the capability. Only RAM, Frame, or DevFrame can be sent. If it is the case, the capability is serialized, sent, and forged on the other side.

H. The init process on the client side put the cap into the init LMP channel. The capability is grabbed by the client.

I. The receiver-side init sends the ACK back to the sender-side init.

J. The sender-side init sends the ACK back to the sender.

6.6 Performance Evaluation

Instead of measuring the performance of UMP channels, we measure the performance of UMP-based Nameservice RPC since the nameserver provides an easy way to bind processes on different cores. The nameservice RPC is just a direct wrapper around the UMP channel. We think an additional function call should not add significant overhead to the performance and therefore the result in Section 11.6 should be valid.

6.7 Limitations and Known Issues

6.7.1 Ringbuffer has a fixed size of a base page

At an early stage of designing the ringbuffer, we decide to use the whole page for a ringbuffer, since it is the minimal size of a frame. However, in many use cases, we put two ringbuffers next to each other to form a bidirectional UMP channel, or four ringbuffers as bidirectional RPC channels (discussed in the RPC section). In these cases, those ringbuffers can share the same frame and therefore reduce the memory footprint.

6.7.2 Ringbuffer uses malloc to return the message payload, which can introduce non-trivial overhead

Using malloc allows passing arbitrary large messages conveniently. However, at a later stage of development, we find that such a design choice may significantly harm the performance of the ringbuffer, especially for high-performance applications like the ethernet.

Chapter 7

Remote Procedure Call (RPC)

In a microkernel OS, the remote procedure call (RPC) infrastructure lay the foundation for all kinds of OS services. In this section, we discuss how we build Ozone's RPC system upon the LMP and UMP systems.

7.1 Unifying LMP and UMP Channels: AOS Channel

At the early stage of development, we implemented RPC purely on the LMP channel, which is sufficient for applications running on a single core. When extending the system to multiple cores, we extended the RPC subsystem by letting init to forward the message.

However, as more components are added to the system, such as drivers, the filesystem and the nameserver, such implementation became insufficient. Considering the performance, when two user-level programs on different cores want to communicate, the message needs to go through LMP, UMP between inits, and then another LMP. Two context switches are involved, as well as multiple data serializations and deserializations, which introduce a lot of overhead.

When making a RPC to another process on the same core, LMP is preferred from the performance perspective. There must be at least one context switch from the sender to the receiver before the message can be handled. In this case, switching in and out the kernel in the LMP call is not an overhead. Furthermore, the scheduler invocation helps reduce waiting time before the message is received on the other side. In contrast, when passing messages to another core, direct UMP is preferred. There is no overhead for context switching at all if no capability is passed.

To provide a unified interface and reuse code, we design AOS channel (`struct aos_chan`). An AOS channel can be initialized to either LMP or UMP, while the low level operations are hidden from the user. Messages are automatically serialized to and deserialized from the LMP message format in Section 5.1 or the UMP format in Section . The 8-bit identifier in both formats can be used to distinguish different calls on the same RPC channel. The generic LMP/UMP message handlers call back the user-specified handler in the following signature:

```
typedef errval_t (*aos_chan_handler_t)(void *arg, rpc_identifier_t identifier,
                                        void *in_payload, size_t in_size,
                                        struct capref in_cap, void **out_payload,
                                        size_t *out_size, struct capref *out_cap,
                                        bool *free_out_payload, bool *re_register);
```

where `arg` is a user-specified argument when registering the handler.

7.2 Building RPC Channel on AOS Channel

On top of the unified AOS channel, we build the RPC interface. The caller makes a blocking RPC call by sending the calling message and listening for the reply. There are two types of reply, ACK and ERR, which is differentiated by the identifier. Given an ACK reply, the RPC call (`aos_chan_call` or `aos_rpc_call`) returns OK and the payload is returned to the caller. If the reply is ERR, the payload is a single error value (`errval_t`). In this case, the error value is read out and return to the caller directly as the return value of the function call, so that the caller does not need to decode the error message manually.

One thing worth pointing out is that, with our current implementation, one **bidirectional** channel supports RPC from **one side**. When the channel is used for RPC calls from process A to process B, A needs to listen on the channel all the time for incoming calling message, which is typically implemented in the event driven style. Messages go to the RPC handler. In contrast, B only listens on the replying way after making the call. The message is directly read out using `lmp/ump_chan_recv`. The same channel cannot be used for RPC from B to A at the same time. Otherwise, there will be racing condition between `lmp/ump_chan_recv` and the RPC handler. Accordingly, two-way RPC requires two bidirectional AOS channels.

7.3 Init Binding and AOS RPCs

Ozone provides basic OS services, such as memory service, spawning service, and so on, to all user programs through the library. All these OS RPCs goes to the init process on the current core.

When a process starts, it needs to bind with init. We create an endpoint in init for each process when spawning the program. The endpoint capability is passed to the user program at a well-known location, namely `cap_initep`. Init starts listening on the endpoint before the process starts, but the channel is yet one way. When a user program starts, it grabs the init endpoint, initializes an LMP channel, and sends the local endpoint through the channel back to init. Init receives the endpoint, completes the binding process, and sends an ACK message back to the process. Upon receiving the ACK message, the user process continue to the program's entry point.

Once the binding is completed, the user program can make AOS RPCs through the library. `libaos` provides wrappers for every available AOS RPC, which encapsulate the actual calls to the RPC function. All RPC calls are blocking.

7.4 URPC Channels Between Inits

In Ozone, every pair of init processes are allowed to make UMP-based RPCs or message passing into each other, subject to some rule to avoid deadlock. As mentioned above, with current implementation, one bidirectional channel supports one-way RPC. Accordingly, two-way RPC requests two bidirectional channels, which take 4 base pages in total. We call such a two-way RPC channel as URPC channel for init, and the 4-page frame as URPC frame.

When the BSP core brings up an APP core, URPC frame is passed to the newly launched CPU driver at a well known location, namely `cap_urpc`. The APP core then sets up a URPC channel with the BSP core. If there are other APP cores already running, the BSP core acts as the coordinator to bind the new core with all running APP core. For example, when core 0 brings up core 2 while core 1 is running, core 0 starts core 2 and setup an URPC channel with it. And then, core 0 allocates another URPC frame, passes it to both core 1 and core 2 on their respective URPC channel, and they setup an URPC channel to each other. The URPC frame, together with the core ID, is serialized into the URPC channel and is forged on the other side.

Chapter 8

Multi-core Support

In this chapter, we describe our work to bring up the second, send a message from core 0 to another core, and spawn a process in that core. First we describe our implementation of bringing up a second core, and conclude with a discussion of sending a message from one core to another. We state that during this milestone, we had already started working on the UMP milestone, therefore we will be delegating a sizeable amount of the explanation to the UMP chapter.

8.1 Booting APP

In this section we describe what we had to do to boot the application cores (APP).
The BSP core is a special core that is brought up during boot, and a special process init is spawned on the bsp core which is responsible for setting up the rest of the system. The BSP core is also responsible for bring up the other APP cores.
The basic idea of booting core is almost as simple as "provide an entry point address, signal the core to wake up and start executing". However, in implementation there is a considerable amount of heavy lifting to be done. The following steps are performed.

1. **Load the boot driver and the cpu driver**: Obtain the binaries for the boot driver and the cpu driver built. We load and relocate these binaries and obtain corresponding memory region objects.
2. **Load init**: Grab the physical memory info corresponding to the binary of init, as we want to spawn init as the first process in the newly spawned core.
3. **Allocate memory**: Allocate the memory required for the cpu driver allocations, and for the kernel stack.
4. **Load URPC frame**: Load the supplied URPC frame into a coredata memreg structure.
5. **Create KCB**: Create the Kernel Control Block by retyping a frame.
6. **Create and fill coredata structure**: Create the coredata structure and fill it with the data obtained in the previous steps
7. **Flush cache**: Flush the coredata and the kernel stack from cache
8. **Call spawn**: Invoke spawn on the coredata

Once these steps are performed, an APP is spawned with the init process, in which in starts executing the app_main function.

8.1.1 Boot all cores

We boot all cores by calling the coreboot function defined in the previous section for all cores in the machine.

8.2 Inter-Core Communication

Inter-Core communication for this milestone was implemented by allocating two shared frames, and initializing a ring buffer in this shared frame. See section 6.1 for details of the ringbuffer. Each core is either the producer or the consumer on a frame. A thread is spawned on the consumer side which waits on messages from the ring buffer and handles them on arrival appropriately.

8.3 Transferring Bootinfo Structure

The bootinfo structure must be sent to the application cores. This is the structure that contains the actual module caps, so it is necessary for other cores to spawn processes. We send the bootinfo struct as a message over the shared channel. The APP core then creates a module cnode and forges all the corresponding module capabilities.

8.4 Memory Transfer

We split up the initial memory received provided to the BSP equally among the 4 cores. The BSP core after adding all RAM capabilities to it's memory manager, allocates a sufficiently large chunk of RAM and sends it to the APP core along with the bootinfo structure. The APP core then forges RAM to this supplied memory region and adds it to its own memory manager. We decided to do it this way and have each have its own memory manager for perfomance reasons. Having to send a request for memory over UMP for every memory request would be slow.

8.5 Spawning a Process in the APP Core

The BSP core sends the name of the binary as the payload of a spawn message. The APP reads it, spawns the process, and returns

8.6 Conclusion

Ozone now has support to launch all (four) cores, and communicate between each other.

Chapter 9

Shell

9.1 The Terminal Server

For any meaningful text-based IO on our system, the Terminal Server is responsible. It is running in user space and handles putchar and getchar calls of other processes. Since there are no virtual terminals on our system, but only one serial terminal instead, the terminal service must also provide the means of managing this resource safely for multiple processes. As we will see this is not a trivial task by any means. In this section we will take a deeper look at this terminal server, its functionality and inner workings.

9.1.1 Execution of the Terminal Server

The Terminal Server on our system resides in the init process and consists of the following components:

Initialisation code for the gic that also runs during the startup of the init process and allows the LPUART to interrupt the driver process (in this case init) instead of constantly polling the device for new input.

Initialisation code for the LPUART device that runs during the startup of the init process and prepares the device for operation later on.

A data structure capturing and storing input in order to prevent input from getting lost while no process is actively listening.

Infrastructure for managing the resource in between multiple processes trying to read and write from/to the terminal.

9.1.2 The General Interrupt Controller

The General Interrupt Controller, or GIC for short, handles the many interrupts that can happen in the system and enables software to register interrupt service routines for the corresponding interrupt types. The terminal server makes use of this functionality in order to reduce the amount of polling that needs to happen in the system. Especially when reading from the terminal, where the actual operations happen rarely (relatively speaking), switching from polling to interrupts makes the system much more efficient.

The setup that happens for the GIC is not very interesting and follows the instructions in the book almost to the letter: first, the memory mapped region of the GIC is mapped into the init process and marked nocache, then the device is initialized with the `gic_dist_init` function, after that an interrupt destination capability is acquired, assigned an interrupt vector and then enabled, which concludes the setup done for the GIC. A later section will go into detail concerning its use for the terminal.

9.1.3 The LPUART

The LPUART driver that has been provided is sufficient for printing and reading single characters to and from the serial connection. Identical to the `gic_dist`, the init binary is linked against the LPUART driver in order to expose the drivers interface and make it possible for init to setup the device. Again the setup done as the book describes it: first the device frame is mapped into local address space again, the device is initialised with the provided LPUART init function and the interrupts for the device are enabled.

This, in combination with the GIC above, is the gist of the setup needed for the terminal to work and thus it is able to perform its basic functions.

9.1.4 Printing

Since the LPUART is initialised now, a simple putchar call can now be executed. The printing direction is actually quite simple and most of its complexity comes from the thread safety that will be explained in a later section.

Fundamentally, when a process tries to print something to the terminal, all that needs to happen is that the actual text is transferred to the terminal server where it is then fed to the LPUART and appears on any connected serial console. The initial rpc infrastructure that was provided with the template offered the `aos_rpc_serial_putchar` function to that end, but we have made an improvement to this RPC call: Since `libc` rarely prints only single characters, we added another RPC call for optimising this a bit, I have added the `aos_rpc_serial_puts` function. As its name applies, this function prints a string instead of a single character and thus only a single rpc call needs to happen, even for large strings being printed.

In reality there are multiple processes on the system that could potentially get in each others way. In addition to the user-level processes running on the system, there is also the CPU driver that is also running continuously. Especially noticeable during the booting process, where some user level processes are starting up on the bsp core, producing a lot of output, and the CPU driver on different cores, also producing a lot of output, neither output is readable since the two processes write into the data register of the LPUART at the same time. In order to prevent this, the kernel's print function, `printk`, makes use of a global spinlock that is taken before printing a message, and then released again when printing has completed. At first I intended to do the same for the user space terminal server, however I came to realize that this approach is flawed. Since the CPU driver has its own driver for the LPUART, it is still possible for two writes to happen concurrently, one in the CPU driver and one in a user space process. These would not know of each other and thus cannot provide any semblance of mutual exclusion.

The solution I have implemented for this problem is that the terminal server shares the lock of the CPU driver. In the CPU driver, the lock is a member of a global data structure and the CPU driver even offers an invocation to get its physical address. Making use of this invocation of the `cap_kernel` capability, the memory region of this global data structure is mapped into the init process where the terminal resides and the `spinlock_t` is used for synchronising any printing that happens both in the CPU driver and any user space process.

9.1.5 The Input Buffer

Handling keyboard input is a different story however. Since key presses tend to happen asynchronously, any process currently accepting input does not necessarily consume it instantly. This leaves two possibilities for the handling of key presses. Either they are passed to the process as an interrupt, or they are buffered until the process is ready to consume them. I have implemented the latter and added a buffer to the terminal server that is filled when characters arrive through the LPUART consumed when a process reads from `stdin`. This approach is much easier to implement, since only on interrupt needs to be set up and no additional infrastructure in the processes themselves is needed. However, this approach also does not allow for the process to be interrupted using a key press such as the well known 'ctrl-c'.

9.1.6 Multiplexing of stdin

The reader may wonder at this point, how the terminal knows what process currently can accept input. This is a great question and its solutions are far from trivial. First there is one very important rule that needs to be followed:

- There can only be one process accepting input at the same time.

Failure to comply with this rule will result in strange effects visible where ever characters are read from stdin, because it is next to impossible to define who is supposed to receive a character other than a simple first-come-first-serve approach, which obviously will result in an interesting distribution of the input. Imagine two processes simultaneously executing a getchar calls until a newline is reached. For every character that arrives through the LPUART, a destination needs to be chosen somehow. If the destination varies while accepting input, the result is a highly confusing interface where a user cannot simply type to their heart's content into the terminal, since the input may not be guaranteed to arrive in one piece at one specific process. Therefore some mechanism must be in place that clearly and transparently defines where the input is headed and enforces the rule above.

Unlike the fancy operating systems of the modern day, our system does not have a collection of virtual terminals and console windows wrapped in an intuitive GUI, but rather only has a single serial connection that can receive input from some other device. So instead of setting up a priority in between systems for given a 'focus' to a process, in our case it suffices to manage the input resource using a stack-like data structure.

This stack contains terminal states that identify different processes for the terminal server. A terminal state is a data structure in the address space of the terminal server, whose address is passed to other processes as an unforgeable token. Since other processes do not have access to the terminal servers address space, they cannot modify their place in the stack or any potential attributes that its terminal state may have.

When a process tries to read from its stdin, it first needs to pass its terminal state to the terminal server which then checks if this terminal state is at the top of its input management stack. The terminal server responds whether the calling process has access to the stack according to this condition. If the calling process does not have access to the input, it will yield and try again, allowing the CPU to do other work in the meantime. In case it has access to the input, it will start reading characters from the input, which the terminal server will provide, since the callers terminal state is at the top of this stack, or respond with an error stating that there are no characters to be read in which case the caller will yield as well and try again later. When a process is spawned, it can either be spawned without access to stdin, or the spawning process can acquire a new terminal state with access to stdin and pass it to the spawned process, thereby relinquishing its own access to stdin, at least temporarily. When a process is stopped, its terminal state is cleaned up and removed from the stack in the terminal server, thereby moving the states below it up in the order. If the cleaned up terminal state happens to be at the top of the stack, the terminal server will from now on grant read access to the process that is now on top of the stack.

Having now worked with this solution for a while, I can say that it is certainly a usable concept, but not without its problems. The obvious one is that passing pointers around to other processes is insecure. Even though the calling process cannot read any information from the terminals address space, passing a wrong address may cause the terminal server to segfault and stop executing. In an environment where malicious actors may be present, this approach would need a lot more work and additional data structures for being completely secure. In addition, the terminal server does not inherently have a notion of when exactly a process stops running. This is an issue if a process is terminated unexpectedly and cannot release the terminal by itself. The result is a orphaned terminal state that clogs up the data structure of the terminal and prevents the process below it to regain access to the terminal, such as the shell that spawned the crashed process. This problem is also not easily solvable without major changes to the architecture of the terminal server.

9.1.7 Thread Safety

As the section on printing has already hinted, the terminal server is thread-safe, as long as it is used within specification. First of all, I have already mentioned the global printing lock that is also used by the CPU drivers LPUART driver. With the use of this lock, the terminal ensures that write operations are atomic from the time they arrive at the terminal. Since this is a global spinlock, it provides mutual exclusion for CPU driver, user space processes and even user level threads within these processes. Since it provides atomicity at the level of write operations, it is however possible that large messages being printed through printf or the like, are not actually printed atomically. Typically stdout is line buffered and thus printing large, multi-line messages from multiple processes or threads may result in line-interleaved output on the terminal. Even if stdout is not line buffered, there will always be a point where libc will start splitting the message into buffer sized chunks that will then be atomically printed, with possible interleavings in between such chunks.

I find this to be acceptable however, especially since this approach will not block any process for unnecessarily periods of time, only because it waits to print a message while another process is producing a lot of output.

The reading direction is again a bit more nuanced. The terminal only provides mutual exclusion in between processes and not in between different user level threads. This means that the programmer has to manage multiple threads without the help of the terminal. Again I find this an acceptable situation, since there is only a single process in possession of stdin at a time, so that the programmed does not need to worry about other processes, so making use of a simple lock will suffice in the vast majority of cases. In addition, it is not clear what access control policy should be made use of if there are multiple user level threads reading from stdin and thus the programmed would need to intervene anyway.

9.2 The Shell itself

The shell our system serves as an interface to interact the processes running on it and enables the most fundamental operations. Its interface is heavily inspired by, though obviously by far not as powerful as, the typical shell found on Linux systems in the wild, such as bash. I have also integrated some of the proposed functionality from the book as well as some other features. Please note that this shell is intended to be used on an ANSI compliant console that can handle at least some common ANSI escape sequences. Running this shell through a non compliant console will result in a rather interesting output pattern.

9.2.1 Execution of the Shell

This section is a general overview of the lifetime of the shell process. The relevant stages mentioned here will be explained in more detail in the following sections.

The shell is spawned by the init process as the last thing it does before entering the event loop. It is spawned with a terminal state that has access to stdin and thus can read from the terminal. Once the shell process has started and `main` has been called, the shell performs some setup operations to prepare for user interaction. This setup includes setting stdin to unbuffered, which is usually line buffered, because some features of the shell cannot be made use of when line buffered. It also includes the setup of the shell environment that I will detail in a later section.

When the setup is completed the shell enters its command loop. At the beginning of the command loop, the shell will prepare its shell environment to accept a new command line and then read input until a line feed or a carriage return are detected. While doing so, the shell handles escape sequences and echos characters back to the terminal so that the use can see what is typed.

Once the line is done, the shell parses the command line and populates the environments argc and argv values, using the `make_argv` function that I copied from the spawn library (I copied it, so that the shell wouldn't have to be linked against the library, since this functionality is provided by the RPC interface.)

When the state is ready, the shell tries to find a builtin command that matches the first element of argv. If this fails, then the shell tries to spawn a process by its name given in argv's first element. If that fails, the shell will print an error and return to the beginning of the command loop. If the spawning is successful, the shell will wait until it regains access to stdin and then return to the beginning of the command loop.

9.2.2 The Shell Environment

The shell environment is a data structure maintained by the shell and captures the current state of the shell down to the current state of the input. It contains the data structure capturing the shell history, the command buffer that captures the current state of the input and meta information surrounding them as well as the shell. The command buffer is dynamically allocated and will be grown when the input exceeds its length. Growth of this buffer is a simple exponential growth, doubling in length every time it is grown.

In addition, some of the builtin commands require certain information that is usually derived from the shell environment. If these builtins end up spawning another process or running a nested builtin, then the shell environment can be cloned and passed to the executed action in order to preserve the state of the parent shell.

9.2.3 Reading the input

While simply reading character after character from the command line is simple enough, it is not enough for the functionality that the shell offers. For example, the command buffer, where the input is temporarily stored until the command line is parsed and executed, has a fixed size and may not be enough for some input typed by a user. In such a case, the shell needs to reallocate the command buffer and increase its size for more input to fit into the buffer.

Trivially, the shell also needs to react to 'the enter key press', which may be expressed differently, depending on the console connected to the terminal. Since different platforms handle line endings differently, the shell must be flexible and at least support Microsoft's carriage-return newline and UNIX' linefeed line endings.

Additionally, there are escape sequences that may arrive through stdin and need to be handled in some way to prevent echoing them back at the user and exclude them from the command buffer. This means that the escape sequences not only need to be recognised, but also delimited correctly in order to preserve the input typed by the user.

Last but not least, recall that this system only has a single interface with the outside world through the terminal service. This has as consequence, that there may be output produced on the terminal at any time, by some background process, since the printing direction of the terminal does not need any privileges. This can, and given enough time, will cause slight problems with the input handling of the shell, since the user may be in the middle of typing an elaborate command, while the terminal prints a few characters. This will result in an inconsistency between the actual state of the shell, represented in the shell environment, and what the terminal shows, especially, when there are features like command line editing. In order to prevent this, the shell also needs to regularly synchronise its state with the terminal. It does so, by enforcing a state to the terminal by reprinting the current line whenever is has changed.

Since the entire state of the shell is kept in the shell environment structure, the current line of the terminal can easily be constructed from a prefix that contains the current working directory and the command buffer. In order to reduce terminal flickering, produced by repeatedly clearing and rewriting a line, the entire line is prepared in a buffer, prefixed by an escape sequence that moves the cursor to the beginning of the line, and suffixed by escape sequences that clear the rest of the line and set the cursor to the its currently intended position. Since the terminal guarantees atomic writes, this approach works great and manages to keep the state of the terminal consistent with my expectations, with one exception: Once the end of the terminal is reached and the line wraps around, the reprint will wrap around again and the terminal will produce excessive output. since for every single key stroke, the entire line will be reprinted.

9.2.4 Handling of the stdin resource

As the description of the terminal server outlined, a process having access to stdin can pass it along when it spawns another process. This means that the shell is responsible by itself for passing access to stdin to its children. When spawning a child process, the shell acquires a new terminal state with stdin access form the terminal service, thereby giving up its own access. Mechanism is then used by the shell to wait until the child and all its children having access to stdin have exited. The shell can test its own access by querying the terminal server and upon receiving no access, it yields in order to free up CPU time. Once the shell has regained access to stdin (either by the child terminating, or giving up stdin) the shell will stop waiting and resume the command loop.

9.2.5 Executing shell commands

Similar to popular shells like bash, when a command line is not a builtin, the shell tries to spawn a binary of the given name. The shell will request a new terminal state having access to stdin and try to spawn a new process of the specified binary with the newly requested terminal state. If the spawn is successful, the shell will wait to regain access to stdin, and then resume the command loop. If the spawning failed, the shell will release the terminal state and resume the command loop immediately.

9.2.6 Features

This section details the features I have implemented for the shell and briefly explains noteworthy solutions to some problems that these features may have faced during development. Please note that this list only contains features of the shell, such as builtin commands and does mention any other programs that can be invoked via the shell.

Input Tokenisation The shell also features robust input tokenisation by the use of the `make_argv` function from the spawn library. My first attempt at input tokenisation was a self made parser making heavy use of the `strtok` function. While this worked reasonably well, especially using echo and trying to print strings containing white space, the largest shortcoming of this approach becomes visible. Since the Using this, the shell is able to split command lines into white space separated tokens, while respecting quoted strings as single tokens. The result of this tokenisation is written to the `argc` and `argv` fields of the shell environment, exposing them to builtins that may accept arguments.

Command History The shell keeps a history of immutable executed commands, whether successful or unsuccessful. This history currently features a capacity of 1024 commands, after which the oldest will get evicted and forgotten. Every time a command is executed, i.e. enter has been pressed, the shell will add the current command line to a ring-buffer making up the history.

Much like well-known shells, the history is accessible by using the up and down arrows. When the history is activated by pressing the up arrow on the current command line, the shell will read the most recent command from the history and replace the current command line with its contents. In order to preserve the history, no access to the history is ever a write access, of course with the exception of appending new entries to the history. When the command line has been copied, the command a copy of the string is now in the command buffer and can be evaluated like any other command by the shell, appending it once more to the shell history.

Command Line Editing Another feature is command line editing. Against the books claim that this can be a large time sink, the structure of my shell has made it easy to implement this and it took me less than two hours. This feature not only includes the functionality of the backspace key, but also insertions and deletions of character at arbitrary positions in the current command buffer, moving all subsequent characters in the command buffer accordingly and the home and end keys, moving the cursor to the beginning or the end of the line respectively. This can be combined with the command history for a much more convenient use of the shell, especially when typing multiple commands with large portions of equal text. This editing of history entries is non destructive, as the entry is copied from the history into the command buffer before being edited. Once an edited command has been executed, it will be appended to the history, as any command is.

Escape Sequences As the two features above may have led you to believe, the shell supports escape sequences as well. This is done by using the escape character as 'an escape' from the usual character reading loop that happens at the start of the normal command loop. Once an escape character is read, the characters following it are regarded as part of the escape sequence until a termination character is read. ANSI escape sequences are generally terminated by an alphabetical character, or a tilde, so this is the exit condition for the escape sequence parser. Once an escape sequence is completely read, it is passed to the escape sequence handler, which will alter the current shell environment, reflecting the escape sequence.

In my implementation only the most important escape sequences are implemented correctly, since there are a lot of them. The available ones include cursor movement and character deletion. Other escape sequences are captured to prevent strange echos and optionally the user is alerted to the presence of unknown escape sequences.

Spawning of processes When the shell executes a command line that is not a builtin command, then it attempts to spawn a process from a binary of the name specified by the first token on the command line. The shell will prepare for spawning this process by acquiring a new terminal state and defining the core to run the spawned process on. The core is chosen in a round robin fashion, and rotated whenever a new process is successfully spawned.

oncore The oncore builtin spawns a process on the specified core. It is capable of spawning anything that a normal command line can spawn as well, however, unlike a normal command line it cannot execute builtin shell commands. This is because the builtin shell commands are not actually spawned as processes, but rather simply subroutines somewhere in the shell process itself.

cd The cd command lets the shell change its shell environment to reflect a different current working directory. Although this current working directory is not integrated with the libc environment infrastructure, the shell can use this directory to complete relative paths and drastically shorten the arguments needed for file system operations on the command line. Passing no argument to cd will reset the current directory to the shell's home directory, which I set to be `/sdcard/`, the mount point of the SD card, since there is no user that can have a home directory.

pwd The pwd command, as is to be expected, prints the current working directory. This command is not really necessary, since the prefix of the shell command line already displays the current working directory and the shell does not support pipes or output redirection. Never the less, the pwd command is a very common command implemented by almost every shell, so for completeness' sake, I include it here as well.

ls The ls command can list the contents of a directory. If given no arguments, then it lists the contents of the current working directory of the shell, and if given arguments, each of them is interpreted as a directory name, and ls attempts to list their contents sequentially in the order that they appear.

mkdir The mkdir command creates a directory in the file system.

rmdir The rmdir command removes directories from the file system. Every argument it is passed, is interpreted as a directory name and rmdir will try to remove this directory. This command will fail for files and non-empty directories.

rm The rm command removes files from the file system, in contrast to the directories that rmdir command removes. Again every argument passed, is interpreted as its own path identifying a file to be removed. This command will fail if called to remove a directory.

cat The cat command reads the contents of a file and prints them to its `stdout`. If given multiple arguments, it will interpret each argument as a file name and try to read its contents. The output is prefixed with the files name in order to mark the boundaries between the contents of different files.

touch The touch command will touch the files it receives as arguments. This means that it opens the files for writing and then closes them again. The result of this is that the file will exist after the command has executed and can be seen by ls, cat and the like.

write The write command will open a file in write mode and write its arguments (the tokenised command line) to the file separated by a newline. The write command will always start writing at the beginning of the file and truncate its length to the new contents, effectively overwriting the file with its arguments.

append The append command behaves almost identically to the write command, with the added benefit of appending to the file, instead of overwriting it. This command can be used for primitive file editing, since a fully blown file editor would be well outside the scope of this project.

echo The echo command echos its arguments back to the terminal, as one would expect. It prints the tokens it has received as arguments to the terminal, separated by a space, similar to the behaviour of bash' echo command, but unlike the Microsoft PowerShell which prints every token on a separate line. Since the shell is able to tokenise quoted strings, as mentioned above, echo can also be used to print tokens with excessive white space if they are quoted on the command line.

exit The exit command breaks the shell out of the command loop, will eventually terminate the shell. Upon exiting, the shell will release its access to stdin as every process, and thus make space for a parent process to acquire stdin access. In the case where the terminating shell is the only process to have had access to stdin, stdin will be left unused and the system will fall into a non-interactive state, where no input from the terminal will take effect.

kill The kill command can be used to terminate processes via the shell. The kill command will interpret its arguments as PIDs and will try to terminate the processes matching the PID numbers. It will kill the processes in the order the PIDs appear in its arguments.

ps The ps command lists all running processes on the system, excluding init itself, by its PID as well as the name of its executable. ps can be useful to find the PID of a process to kill it.

time The time builtin command approximately measures the time taken to execute a command. This command can be an executable or even another builtin shell command. The time command will measure the time taken from matching against known builtins until the builtin command has completed or a process has been spawned (either successfully or unsuccessfully) and access to stdin is regained. The measurement is taken in microseconds by the systime library. Since the shell is waiting for stdin access before completing the time measurement however, the reported time may not be very accurate as the shell could have been waiting for CPU time for quite some time before being resumed, thus artificially inflating the timing measurement.

The time command can time any arbitrary command line, from other builtin commands, to spawned processes, and even command lines that cannot be executed.

san The san builtin takes a path as argument, and produces a sanitised absolute path, taking the current working directory of the shell into account.

Command Line splitting [ht] Using the ampersand symbol ('&') multiple commands can be written on one command line. They will be split up and started sequentially, with only the last command receiving access to stdin and the others just starting as background processes. This notation supports builtin commands as well as spawn commands at every position.
Example:

```
echo hello & memtest -threads=8 & echo world
```

This will run the first echo, then the memtest command without access to stdin, which is fine because it does not read from stdin, and then the second echo. After that the shell will immediately resume while the memtest process is still running.

Chapter 10

Filesystem

Any real world system requires support for large and persistent storage. The code of an operating system, configurations, files, etc are all stored in disk and are retrieved into memory during boot. Thus, we require a way to interpret all the data stored in disk so they can be obtained and understood after boot. This set of protocols that organize and give meaning to the data stored in disk is what we call a filesystem.
Following tradition, I have implemented a read and write FAT32 filesystem in Ozone. In this section I describe the design details of our implementation. We discuss (and critique) the FAT32 protocol. Then I describe the read interface and discuss the extensions required to implement write support. I then show some benchmarks, discuss my efforts with spawn, and conclude with a discussion of what can be improved.

10.1 FAT32

The FAT32 filesystem is an extension of the FAT (File Allocation Table) filesystem designed by Microsoft. The most important aspects of FAT32 are briefly described below.

1. **Sectors and Clusters** : The disk is divided into sectors (or blocks) which are contiguous regions of data (usually 512 bytes). The disk is accessed in sector granularity. Further, the disk is divided into clusters, which are contiguous regions of sectors (usually 8 to 32). A file or directory may span one or more clusters, and no two files or directories may share the same cluster(s).
2. **Boot Partition Block (BPB)** : The first logical sector (sector 0) contains metadata about the FAT32 volume. Information about parameters such as the starting location of the root directory, number of sectors per cluster, starting location of the FAT etc are contained here. While FAT12 and FAT16 contain the root directory at a fixed location, FAT32 does not. The BPB is primarily important for verifying that the volume contains a well formatted FAT32 partition, and to find the starting location of the root directory.
3. **Directory Entry** : A directory entry (dirent) is a 32 byte entry consisting of metadata that identifies the underlying file or directory. Most importantly it contains the name, identifies whether the entry is a file or a directory, and contains the starting data cluster of the entry.
4. **Files and Directories** : FAT32 is divided into files and directories, and each element must be one or the other. A directory is a collection of 32 byte directory entries and may not contain anything else, while a file may contain any data. All directories must contain "." and ".." as the first two entries, pointing to itself and its parent respectively. On the other hand, files may have an empty data section, or may not have a data section at all (empty file) in which case the first data cluster would point to 0.
5. **File Allocation Table (FAT)** : The FAT, the namesake of the filesystem, is arguably the defining feature of FAT32. The FAT contains an entry for each cluster on disk, and is indexed using a 32 bit

address. Therefore, FAT32 can manage at most $2^3 2$ clusters. The FAT entry for a cluster is simply a 4 byte entry which indicates whether the cluster is free, points to the next cluster, or indicates an EOC (end of cluster). Files and directories spanning multiple clusters must be traversed via the FAT to identify where the next cluster of data is. The EOC entry indicates the final cluster of a file/directory.

While all directories are required to contain "." and ".." entries, the root directory does not. This led to some dirty conditional branches in the code, and could've been avoided if the root directory also followed the standard. The ".." entry for a directory could've simply not existed, leading to cleaner recursive traversal of the filesystem.

10.2 SDHC Interface

10.2.1 Block Driver

A block driver is supplied to read and write from the SD card. The block driver operates using Direct Memory Access (DMA), and therefore requires us to flush the memory region before reading from it or after writing to it.

To loosen this requirement, we created wrappers around the SDHC driver for reading and writing. The read and write wrapper functions map a region of memory as "no-cache" and use that as the intermediary memory region for interfacing with the block driver. Therefore, the caller does not have to worry about flushing the buffer.

Unfortunately, the supplied SDHC reader is considerably slow. Most importantly, some form of caching is absent. This leads to repeated reads from the same sector even if we access it consecutively. A more detailed discussion of this can be seen in 10.7

NOTE: If we want to modify a region in disk that is smaller than the block size, we must read the entire block from disk, modify the small portion, and write the entire modified block back

10.2.2 Extra Challenge Optimizations

I performed one optimization to the block driver. In read_block and write_block, the SDHC driver sets the block len and then reads the block. This caused a lot of redundancy as the block len was being set in every loop, which meant two commands were being sent to the sd card for every read/write. I removed this block len command and instead only do it once during initialization. This roughly doubled the speed of sd reads and writes.

10.2.3 SDHC Initialization

The capability to the SDHC port is retyped into a devframe and provided (see Shell section for more information). This capability is then supplied to sdhc_init to initialize the block driver to the sd card. I found the supplied sdhc_test function very useful to give me confidence that my initialization was correct

10.3 FAT32 Manager

The FAT32 manager handles all filesystem functionality. It is a list of functions combined with a global manager object that maintains the state of the filesystem. In this section, we describe the initialization process and all the structs that I designed to handle functionality.

10.3.1 struct fat32_dirent

The fat32_dirent (henceforth referred to as dirent) struct forms the heart of the filesystem. Almost all fat32 functions either require or return a handle to a dirent.
As mentioned earlier, in FAT32 both files and directories contain the same metadata, with the only differences being that the attribute flag is different, and the data section requiring pre-filled entries for "." and ".." for directories. The dirent structure contains information to uniquely identify a directory entry in disk. In addition to the dirent fields, it also contains *sector* and *sector_offset* fields. These fields identify the logical sector of the parent directory's data cluster where this dirent lies, and the offset into the sector that contains this 32 byte dirent. Note that offset $\bmod 32 = 0$ is an invariant, and is maintained throughout.

10.3.2 struct fat32_handle

The handle simply encapsulates a dirent, and in addition contains an offset field which tracks the current position of the file pointer.

10.3.3 fat32_handle_t

This is just a typedef for a *void**. The caller of functions that return a file are returned a handle to the file of type fat32_handle_t. Underneath, this handle is simply a pointer to a fat32_handle object. As this is a virtual address, this address cannot be interpreted by the caller, and therefore the caller cannot malform these directory entries. This virtual address is supplied when calling fat32 functions that operate on dirents, in which case the filesystem can simply interpret the virtual address in its domain.

struct fat32_manager

The fat32_manager contains the most important BPB metadata like the start of the root directory, start of the data cluster, number of reserved sectors, size of the FAT, etc. These fields are repeatedly used during run time to calculate the index in the FAT table of a cluster and the starting sector of a cluster.
The manager also tracks the list of free clusters. More on this in 10.5.
Finally it also maintains a permanent pointer to the dirent of the root directory. This is used as the starting point of all queries into the filesystem.

10.3.4 Serialization and Deserialization of Directory Entries

As described in the FAT32 specification, a 32 byte directory entry has a specific layout by which it is stored on disk. I have written a serialization function which given a directory entry, serializes it into a 32 byte buffer, and conversely a deserializer which given a buffer, parses it and produces a fat32_dirent object.
As dirents are created when a particular path is requested by a user and not stored except in the handle returned to the user, when close or closedir are called, these dirents are freed. However, note that if the root directory is opened, the fat32_manager's persistent root_directory entry is supplied to the caller. Therefore, if the caller attempts to close the root directory, we must NOT free the underlying dirent, as we then risk destroying our filesystem (I did run into this bug and spent hours trying to solve it till I realized I was freeing the root directory).

10.3.5 FAT32 Initialization

When initializing the filesystem during boot time, init on the BSP core does the following in order.

1. Map the DevFrame capability to the SDHC port, and initialize the block driver
2. Read the Boot Partition Block (BPB) of the FAT32 formatted SD card and verify the first three bytes. If verification fails, abort filesystem initialization
3. From the BPB, read the relevant fields to initialize the FAT32 manager

4. Create a root directory entry in the FAT32 manager, and initialize it to the first cluster of the root directory
5. Set root directory's path to be the supplied mount argument ("/sdcard" usually)
6. Traverse the FAT and determine the list of all free clusters (see 10.5)

As the FAT32 manager is initialized using parameters in the BPB, the implementation can support arbitrarily formatted FAT32 volumes. However, it only supports one partition.

10.4 Read

In this section I describe the implementation of the filesystem's read functionality. First I describe the motivation behind the design, then describe the design itself.

10.4.1 Motivation

It is important to mention that before I began to write code I spent considerable amount of time analyzing the workflow of the filesystem. There are over 15 functions, therefore I wanted to minimize duplication. What do each of the functions do? How can I abstract away the underlying functionality? Finally, how can I reuse functionality?

From a quick glance of the interface, we notice immediately that the interface can be divided into file operations and directory operations. However, we also observe that most "file" functions also posses a directory analogue. But, files and directories are really the same thing in FAT32, and only matter when it comes to actually writing data into them or reading from them. Therefore I naturally decided to group functionality in the following way.

1. open, opendir
2. create, mkdir
3. rm, rmdir
4. close, closedir
5. read, readdir
6. write

We do not consider close and closedir further as their functionality is trivially freeing up the underlying dirent structure

As seen in Figure 10.1 the filesystem is a tree of directory entries, where each node is a directory, and each leaf is either a directory or a file. I further observed that the filesystem function can be exclusively split into the following two categories.

1. Given a path, find a dirent and return it : open, opendir
2. Given a path, modify a dirent in the location : create, mkdir, rm, rmdir
3. Given a handle, modify the dirent pointed by the handle : read, readdir, write

However, it is clear that category 1 and 2 are similar to each except for the operation being performed. Much of their functionality can be abstracted out and reused, with simply the final operation requiring a change. Thus, I designed the following algorithm for category 1 and 2.

1. Given a path, recursively traverse the tree and find the location
2. If final dirent in the path does not exist, and CREATE_IF_NOT_EXISTS flag is set, create this dirent.
3. Return the found dirent

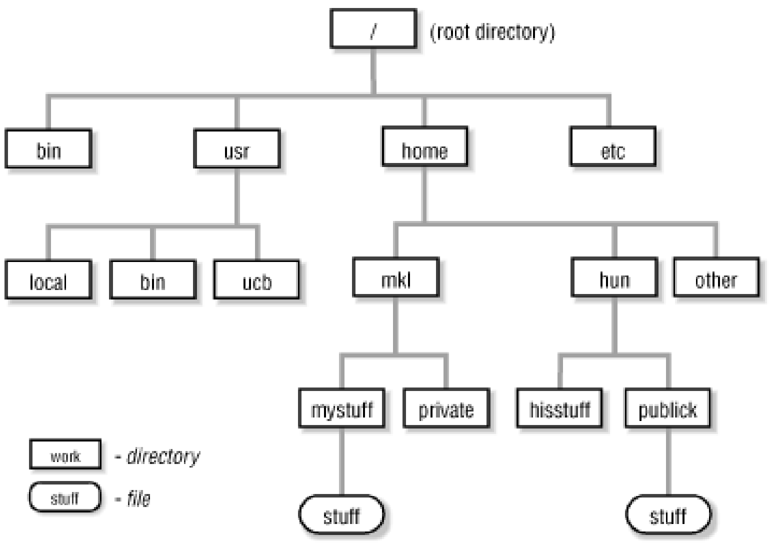

Figure 10.1: Tree representation of a filesystem [1]

```c
/**
 * @brief A super function that does a variety of things based on the given dir.
 * @param dir           The directory we are searching in.
 * @param name          The name of the dirent to search for, is ignored if find_empty.
 * @param find_empty    If set, instead of finding the first dirent that matches the name,
 *                      it finds the first empty dirent.
 * @param retdir        If find_empty is set, this is ignored. Otherwise, return the found
 *                      dirent if found.
 * @param retsector     If find_empty is set, return the sector of the found empty space.
 *                      Otherwise, ignored.
 * @param retoffset     If find_empty is set, return the offset into the sector of the
    found
 *                      empty space. Otherwise, ignored.
 * @param retcluster    If find_empty is set, return the cluster of the found empty space.
 *                      Otherwise, ignored.
 * @return FS_ERR_NOTFOUND or SYS_ERR_OK
 */
static errval_t find_in_directory(struct fat32_dirent *dir, const char *name,
                                  bool find_empty, struct fat32_dirent **retdir,
                                  int *retsector, int *retoffset, int *retcluster) {
```

Figure 10.2: The super function

We immediately observe that all the category 1 and 2 functions can be now implemented trivially with a call to this function and the appropriate parameters. Rm and rmdir can find the required directory using the above algorithm, and then zero the found sector of the dirent.

10.4.2 Find or Create Dirent

The algorithm described in the previous section is the backbone of the entire FAT32 implementation. It is called *find_in_directory* (henceforth referred to as the super function) and virtually every FAT32 function passes through it. Figure 10.2 provides a view of its header. It is indeed a complicated function, but it makes life easier in a number of ways.

- The super function was used to implement a general search function (Figure 10.3). This find function decomposes the given path into a number of disjoint directories, and for each such directory in the path, it iteratively calls the super function to find the next directory till we reach the end of the path.

[1]https://docstore.mik.ua/orelly/unix3/upt/ch01_14.html

```c
/**
 * @brief Given the current directory and a relative path, search the directory for the
 *        resulting path, and create the last step dirent if needed.
 * @param curr    Current directory.
 * @param path    Path from current directory.
 * @param CREATE_IF_NOT_EXIST  Create the last level dirent if it does not exist.
 * @param Attr    Attribute of the dirent we are creating.
 * @param retent  Return a pointer to the directory entry found/created
 * @return  SYS_ERR_OK or FS_ERR_NOTDIR or FS_ERR_NOTFOUND
 */
static errval_t search_dirent(struct fat32_dirent *curr, const char *path,
                              bool CREATE_IF_NOT_EXIST, uint8_t Attr,
                              struct fat32_dirent **retent) {
```

Figure 10.3: The search function

- The search function implemented using the super function was also used to implement creation. A minor modification was made such that the general search function also takes a boolean which indicates whether a dirent should be created or not. If the last dirent in the path is not found, and the boolean flag is not sent, then a new dirent of the supplied attribute (file or directory) is created (see 10.5).
- It is helpful also for writing into files/extending directories. I will revisit this in section 10.5

10.4.3 Read From File

Reading from a file is straightforward. Starting from the file position in the handle, in every iteration, either SDHC_BLOCK_SIZE bytes are read into the buffer, or user provided bytes is read, whichever is smaller. Then, bytes is decremented by amount read, and the next sector is read till bytes is 0. A noteworthy point is that it is not possible to merely increment sectors indefinitely from the start sector. As soon as the cluster boundary is crossed, the next cluster and sector is read and calculated from the FAT. Therefore, every read requires $\frac{bytes}{blocksize} + \frac{bytes}{clustersize}$ block reads from the sd card.

10.4.4 Read Next Dir

The position variable in the handle is interpreted differently based on the whether the dirent is a directory or a file. For a file, as mentioned in the previous section, the offset is simply the byte offset into the file. However, for a direnent, I intepret it as the directory index. Therefore, the byte offset would be $offset * 32$. Therefore, when reading a directory readdir reads the next dirent at $offset * 32$ position. The reason for this design choice is that it is less likely to have bugs because by definition the actual offset is guaranteed to be divisible by 32, and therefore offset increments are no longer a source of error.

10.5 Writes

In this section, I describe the underlying mechanism of writes, and how I extend a dirent to another cluster. Furthermore, I also discuss how deletion of a directory is implemented.

10.5.1 Cluster Chains and Extension

Every file or directory in FAT32 is a chain of clusters. The first cluster of data can be found in the dirent, and to get the subsequent clusters we recursively index the FAT. The FAT entry of an index contains one of the following: a 0 indicating a free cluster, an EOC marker indicating the end of the cluster chain, or a 32bit integer pointing to the next cluster in the chain.

When a write requires more space than exists in the data cluster of the dirent, we need to extend the cluster of the dirent. We observe that extending is merely allocating a cluster, writing EOF to the FAT of this newly allocated cluster, and writing the allocated cluster number into the FAT of previous last cluster.

How do we allocate a cluster?

10.5.2 Cluster Allocation and The Free Cluster List

It is necessary to maintain a list of free clusters that we can allocate on demand to grow files and directories. The free cluster list is a linked list that is initialized on boot time and is populated with a fixed number of free clusters. The reason for this is that the FAT is considerably large, and traversing the entire FAT would make boot unnecessarily longer. I instead refill the free cluster list on demand. Every refill of the free cluster list tries to add at most K free clusters to the list. After calling refill, the last traversed position is remembered, and the next call of refill simply resumes from there.

NOTE: Refill assumes that once a cluster is initially added to the free cluster list, it is never "lost". That is, every allocated cluster either is occupied, or is re-added to the free cluster list after it is freed. This invariant is maintained by the filesystem functions.

10.5.3 Write to File

Write to file follows a similar loop as section 10.4.3. Every iteration, the minimum of SDHC_BLOCK_SIZE or remaining bytes is written. Cluster boundaries are similarly handled. The primary difference is that if the loop reaches EOC, a new cluster is allocated and the file is extended with the new cluster, and write proceeds.

Finally, after write is completed, the new size of the file is written back into the file's dirent.

10.5.4 Mkdir

First the given path is resolved. If the directory is not found we error and return immediately.
Once the function has the directory in which we want to create a new directory, the function then needs to find the first empty 32 byte "slot". If there are no such slots, the functions allocates a new cluster and extends the directory. The search function in Figure 10.3 is once again useful here, as it also finds the first empty slot in a directory.
With the sector and offset of the empty slot, the function initializes a dirent object corresponding to the directory to be created, serializes it into the sector and offset read from SD and writes it back.

10.5.5 Deletion

The function finds the file to be deleted. Then, it traverses the cluster chain of the FAT to identify all the occupied clusters. It writes 0 (free cluster) to the FAT of all these clusters, and adds them back to the free cluster list.
Finally, the dirent in the parent directory needs to be freed. However this has to be done carefully, because we either write a 0x0 or a 0xE5 to the first byte respectively depending on whether deleted directory is the last directory, or if there may possibly be other directories after the deleted one.
Why does it matter? Why not just write 0xE5 and call it a day? The reason is performance. When a 0x0 is encountered, we may immediately terminate lookup as there is a guarantee that there are no more dirents after that. So if we exclusively write a 0xE5, after a number of deletions lookups would take a very large amount of time to terminate as most of them waste time looking through large regions of empty spaces.
The general delete function is seen in Figure 10.4.

Rm directly calls the delete function. While Rmdir also additionally checks that the directory is empty before deleting it.

```c
static errval_t delete_dirent(struct fat32_dirent *dir) {
    errval_t err;

    //cannot delete root directory
    if(dir->parent == NULL)
        return FS_ERR_ROOT_DELETE;

    if(dir->is_dir) {
        bool is_last;
        CHECK_ERR(is_last_dirent(dir->FstCluster, 1, &is_last), "");
        if(!is_last)
            return FS_ERR_NOTEMPTY;
    }

    CHECK_ERR(burn_cluster_chain(dir->FstCluster), "");

    // Check if dir is the last directory entry of the directory we are deleting it from
    int parent_sector = FIRST_SECTOR_OF_CLUSTER(dir->parent->FstCluster);
    int offset = (dir->sector + dir->sector_offset - parent_sector)/32;
    bool is_last_in_parent;
    CHECK_ERR(is_last_dirent(dir->parent->FstCluster, offset, &is_last_in_parent), "");

    uint8_t data[SDHC_BLOCK_SIZE];
    CHECK_ERR(sd_read_sector(dir->sector, data), "");
    data[dir->sector_offset] = is_last_in_parent ? DIR_ALL_FREE : DIR_FREE;
    CHECK_ERR(sd_write_sector(dir->sector, data), "");

    return SYS_ERR_OK;
}
```

Figure 10.4: The delete function

10.6 RPC

The filesystem is implemented in the init process on core 0 (BSP core). All processes (including init) are required to call filesystem_init() before being able to interface with the filesystem. If the calling process is init on core 0, filesystem_init() links the libc functions to the fat32 functions, otherwise filesystem_init() links them to the corresponding remote procedure calls (RPCs).
The filesystem RPCs and handlers together are responsible for sending a filesystem request to init0, getting the response from init0, and returning it to the calling process.
Each filesystem function has a separate RPC handler. It is important to mention that the RPC handler is synchronous, therefore subsequent RPCs must wait for the previous one to finish. This guarantees thread safety, as no two threads or processes can ever be in a filesystem function at the same time.

10.7 Evaluation

In this section I present the evaluation of three elements namely: SD Read and Write, File Read and mkdir.

10.7.1 Setup

I timed the performance of the functions using the systime functions. To evaluate reads and writes, I averaged the time over reading/writing to a very large text file (32KB).
For testing fread and mkdir I averaged it over 10 independent trials.

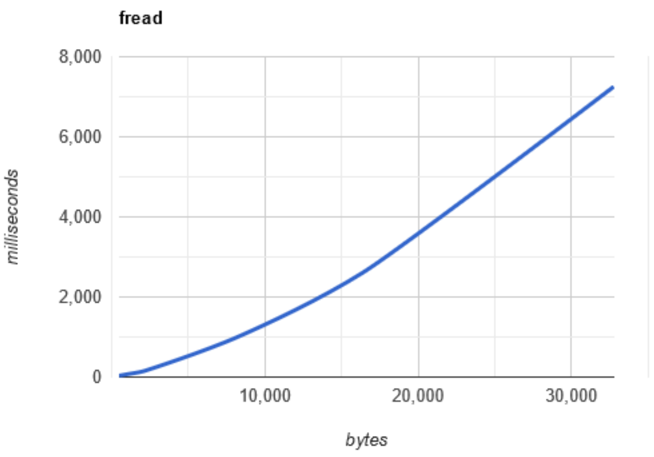

Figure 10.5: File read benchmark

10.7.2 SD Read Write Bandwidth

In Init

I measured an sd read time of 320ms while reading the boot partition block. This would suggest a bandwidth of 1600 bytes per second.

In RPC Handler

I measured an sd read and write time of 36ms on average for reading/writing a block of size 512 bytes. This implies a bandwidth of 14222 bytes per second.

Discussion

I am not exactly aware why this difference in read times exist. One possible explanation I have for this is that the RPC upcall may disable something like interrupts, which would lead to sdhc_read and sdhc_write no longer being blocking. This explanation aligns well with the results I present in Section 10.8

10.7.3 File Read

Figure 10.5 shows the results of the file read benchmark for sizes 256B, 512B, 1KB, 2KB, 4KB, 8KB, 16KB and 32KB. We observe that the graph steadily grows, implying that the latency for reading bytes does not grow linearly. This is correct and aligns with my expectations as presented in a formula in section 10.4.3. There are two primary bottlenecks here.

1. Everytime we cross a cluster boundary (4KB), we have to also fetch from the FAT the corresponding next cluster entry. Therefore, latency increases for every cluster boundary that has to be crossed
2. The libc fread function surprisingly splits up the read into multiple 1024Kb reads. Therefore, every read would end up having to traverse the FAT chain repeatedly. This latency can be solved by tracking the current cluster and sector of the file handle.

10.7.4 mkdir

Figure 10.6 shows the results of the mkdir benchmark for levels 0 to 6. Level stands for the depth of the directory being created. We see a linear increase in latencies, which is in expectation as the deeper you

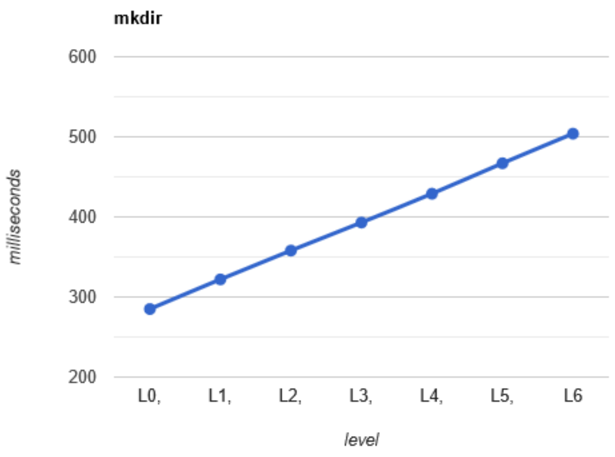

Figure 10.6: mkdir benchmark

go, the more blocks you have to read to reach the required directory

10.8 Spawning

In this section, I describe my attempts to implement process spawning (or a lack thereof) in the filesystem. I will mention that I was unsuccessful and am unsure why, therefore I have described the countless hours I have spent debugging in this section.
With a working filesystem, process spawning should be simple. Simply read the binary from the filesystem, marshall it into a spawninfo structure, and pass it to the spawn function in spawn.c. Modify the elf mapping functionality to use the supplied virtual address of the binary instead of mapping it from a cnode_module capability. Then everything should take care of itself.
However this did not work and segfaulted when trying to get ".got" location. Upon close inspection of the binary, it turned out that some characters in regular intervals from the 500 or so byte onwards would mismatch the correct binary. So in conclusion, the reads are incorrect.
Therefore, my first assumption is that I wasn't using the sdhc driver correctly. However, that was not the problem because I am mapping a page no-cache, moving data into it, and using the no-cache mapped page for the actual driver interface.
Finally I thought that the very fast SD read was the reason, and maybe it wasn't blocking for some reason. I introduced usleep delays after an sdhc read of upto 450ms. While this seemed to have improved the accuracy as more bytes were now clean, after 250ms of delay, for all delays every byte after byte 4068 would mismatch. At this point I was no longer sure what the problem was.

Chapter 11

Nameserver

In a microkernel OS, the nameservice is also an essential component for inter-process communication. The nameserver acts as a bridge between service providers and the user programs. A server registers the service it provides with a unique name, while a client obtains an IPC channel to the server by looking up the service.

A large amount of work of this individual project is not in the nameserver process or the library support, but in the underlying RPC system, which in turn relies on the LMP and the UMP systems. We have made significant refactoring to all these IPC systems as part of this project. For example, at the early stage of development, Ozone only supported RPC on LMP channels, while inter-core messages are forwarded by the init process. To support more efficient communication between arbitrary processes, we extend the UMP system and make it available to all processes through the library. Also, we create the unified AOS channel to hide the underlying communication protocol from the user. All these works are mostly discussed in the previous sections. In this section, we focus on the implementation of the library support of nameservice and the nameserver process.

11.1 System Architecture

The nameservice is jointly provided by library code, the nameserver process, and init. The library support for service providers (servers) and users (clients) is distinct. Figure 11.1 shows the architecture of the nameservice system. The bold lines show low-level connections between processes. The dashed lines show operations, which we will discuss in detail in the following sections. Although we show server- and client-side library support in two blocks, they are essentially in the same library, namely `libaos`. A process can act as a server and a client at the same time. The channel between the user process and the nameserver (NS Channel) is shared by both library components.

11.2 Binding to Nameserver

The nameserver itself is a process running on the BSP core. User programs need to have a way to bind the nameserver at bootstrapping. Dashed lines A to C in Figure 11.1 shows the binding process between the server and the nameserver. The same procedure applies to the client process, which is omitted in the image for brevity. Specifically, the binding process takes 3 steps:

A. The process requests binding to the nameserver through an init RPC call. The init acts as the coordinator to set up the UMP channel (NS Channels in the plot) between the process and the nameserver. As discussed in the UMP section, it involves creating a shared frame and passing it to both sides.

B. The frame capability is returned to the process as the return of the RPC call.

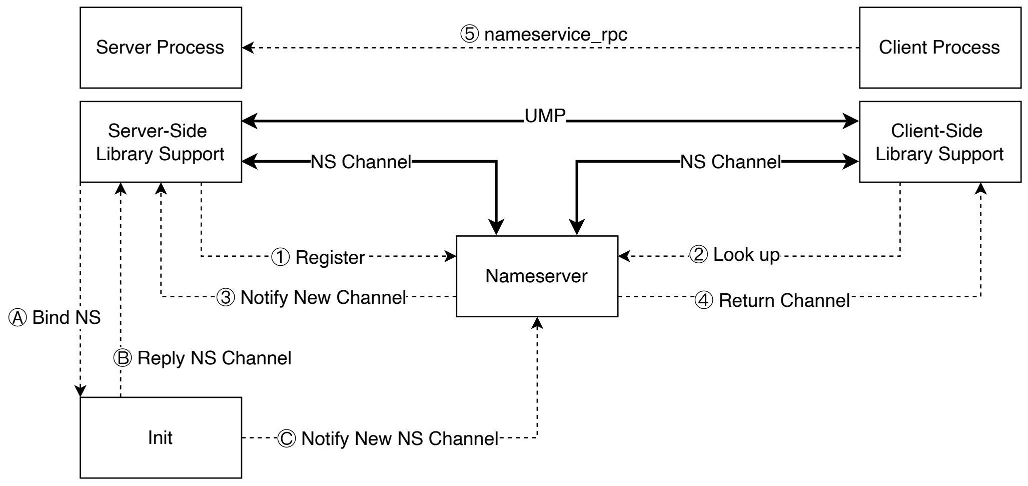

Figure 11.1: Architecture and operations of the nameservice system.

C. The frame capability is passed from init to the nameserver through an LMP channel that is set up when the nameserver starts. If the init process is not on the BSP core, the frame is serialized, sent, reconstructed in init.0, and then passed to the nameserver in LMP.

Each NS Channel includes two UMP channels next to each other, packed into the single shared frame capability. The first channel is for the user process to make RPCs to the nameserver, such as registering, deregistering, looking up, etc. The second channel is for the nameserver to notify the server-side library about newly established channels. As mentioned above, a process can act as a server and a client at the same time. The NS Channel is shared by the server- and client-side library. Therefore, the notification channel is created upon binding even if the process only acts as a client and never uses it.

11.3 Nameservice Operations

Dashed line 1 to 4 in Figure 11.1 demonstrates the nameservice operations. All these operations are RPCs on the NS Channels. Specifically,

1. The server registers a service with `nameservice_register`. Nameserver checks for name conflict. If the name is unique among living services, the service is registered. Otherwise, an error is returned. Upon success, the server-side library also created a local record of the service (discussed in the next section) to store the user-specified handler and argument.

2. The client looks up the service with its name using `nameservice_lookup`. Nameserver looks up the service. If no such service is found, an error is returned to the client. Otherwise, the nameserver acts as the coordinator to set up the direct UMP channel between the server and the client.

3. Nameserver allocates a frame and passed it to the server through the notification channel. The frame is passed with the indirect UMP capability transfer mechanism mentioned in the previous section. The server-side library opens the UMP channel between the server and the client using the frame. In case there are multiple services registered by the server, the notification message contains the service name that gets looked up. The server looks up the local record and starts listening on the channel.

4. The frame is returned to the client as the return payload of the RPC call. It's also passed by the indirect capability transfer mechanism at low level. The client-side library opens the UMP channel between the server and the client using the frame.

5. After the UMP channel is set up, the client and the server communicate directly in the user space, without the interference from the nameserver. The `nameservice_rpc` call is basically a wrapper of the underlying AOS channel (UMP channel) RPC. On the server side, an RPC handler in the

library receives the incoming message and dispatches it to the user-specified handler.

Similarly, deregistering involves an RPC to the nameserver and deleting the local record of the server. The nameserver checks if the process is the same as the one that registered the service. If not, the operation is rejected and an error is returned to the caller.

Nameservice enumeration is a single RPC to the nameserver. We support prefix matching in our current implementation.

11.4 Data Structures

In addition to registering services with the nameserver, the server-side library also maintains a local copy of registered services from the current process in a doubly-linked list, where each node stores the name, the handler pointer, and the user-specified argument. This data structure is passed to the UMP handler to dispatch events to the user-specified handler.

Both the server- and the client-side library maintains a doubly-linked list of living connections, which essentially holds the instances of the underlying AOS channels. Connections are kept alive even if the service is withdrawn.

As for the nameserver process, it maintains a doubly-linked list of processes that are bound to the nameserver, holding the RPC channel to listen on and the notification channel. For registered services, a red-black tree is used to manage them. Nodes in the RB tree are indexed by the service name. Looking up a service is an exact matching, which takes $O(m \log n)$ time, where m is the maximum service name (maximum complexity to compare two names) and n is the number of registered services. Enumerating services takes $O(mn)$ time since every node is compared against the prefix.

11.5 Dead service removal

Nameserver records PIDs of each server. When a process exits, init notifies the nameserver with its PID, and all services registered by the process are deregistered.

11.6 Evaluation

The nameservice acts as the backbone of other functionalities in Ozone such as the ethernet driver, which means it needs to be functional to support those services. In addition, we implement several test programs to evaluate the nameserver functionality. For the full list of them, please refer to the appendix. Here we discuss one of the test program: `nameservicetest` and the result from `nametime`.

`nameservicetest` is an extended version of the provided test program that stress tests all the functionalities of the nameservice. It starts by registering a service, and it spawns 8 clients on all the cores in round robin. All clients try to bind to the nameserver and look up the service, which requires robust concurrency handling. In our evaluation, it works correctly. Each client sends a message and a frame capability with a text written inside. The server receives the message and the frame, prints them, and replies a message as well as another frame. The client prints the reply message and the frame.

`nametime` is used to evaluate the performance of the nameservice RPC. It acts as a server or a client depending on the program argument. When the argument is "server", it registers a service and waits for the incoming calls. Otherwise, it acts as a client that makes RPC calls to the service. Timestamps are encoded in the message payload so that we can measure the one-way time and the round-trip time.

Figure 11.2 shows the one-way time from the client to the server with different payload sizes (no capability is passed). The server and the client run on different cores connected by a UMP-based nameservice channel. Generally, the one-way time increases as the payload size increases, which is expected since more bytes are copied and more cachelines are synchronized to the other core passing the payload. When the payload size is 64 bytes, the one-way time stays around 30us. When approaching 4096 bytes, some data points lie around 100us while there is increasing number of data points that goes beyond

300us. A possible explanation for those data points is that the scheduler is moving to other process so that the UMP messages are not handled instantly.

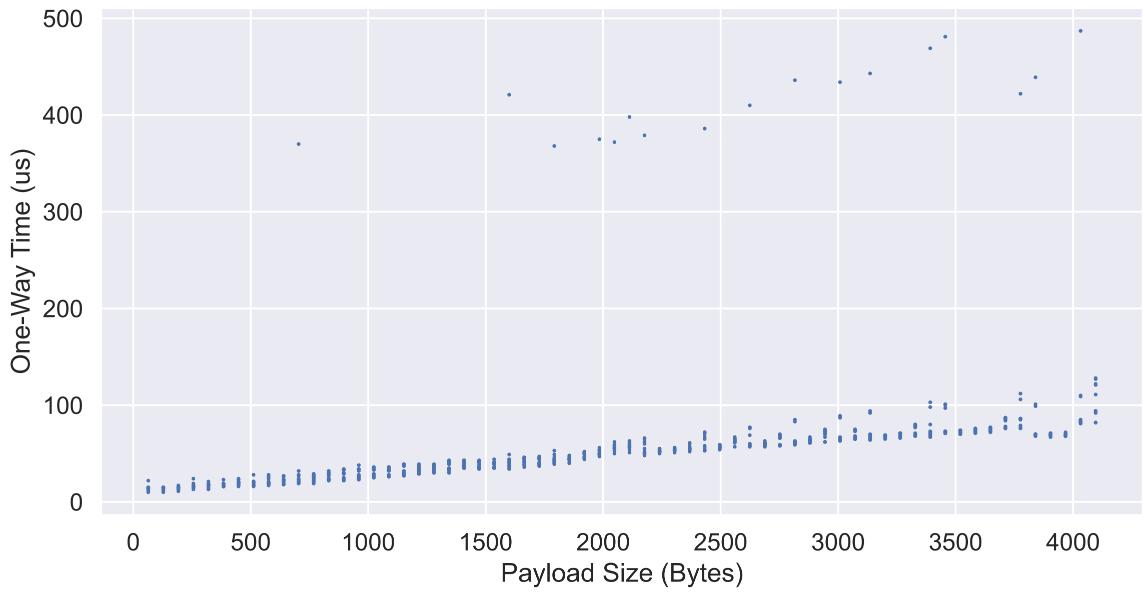

Figure 11.2: One-way time of a nameservice RPC with different payload sizes. No capability is passed. The server and the client run on different cores connected by UMP-based nameservice channel. One outlier: 2235us when the payload size is 4096, is omitted for readability.

Figure 11.3 shows the round-trip time of the nameservice RPCs. One thing we immediately notice is that most of the data points stay above 250us, which is much more than doubling the corresponding one-way time. One possible explanation is that most of the round-trip time is large enough to make the client get context switched out before the message comes back. In fact, data points in Figure 11.3 lie around 100us, 450us, and 750us, which have a roughly 300 or 350us interval. If one time piece of the scheduler is about this length, the data would make sense. But overall, we think a few hundred microseconds are an acceptable performance of the RPC call.

11.7 Limitations and Known Issues

11.7.1 Dead services are removed while dead channels are not

Dead services are automatically removed by the nameserver. However, in the library of nameservice, channels are not automatically removed. One main reason is that nameservice RPC channels are opaque pointers (`nameservice_chan_t`) to the user, and the library has no control over it. If the channel is released in the library while the user is still holding the dangling pointer, an unexpected error can occur.

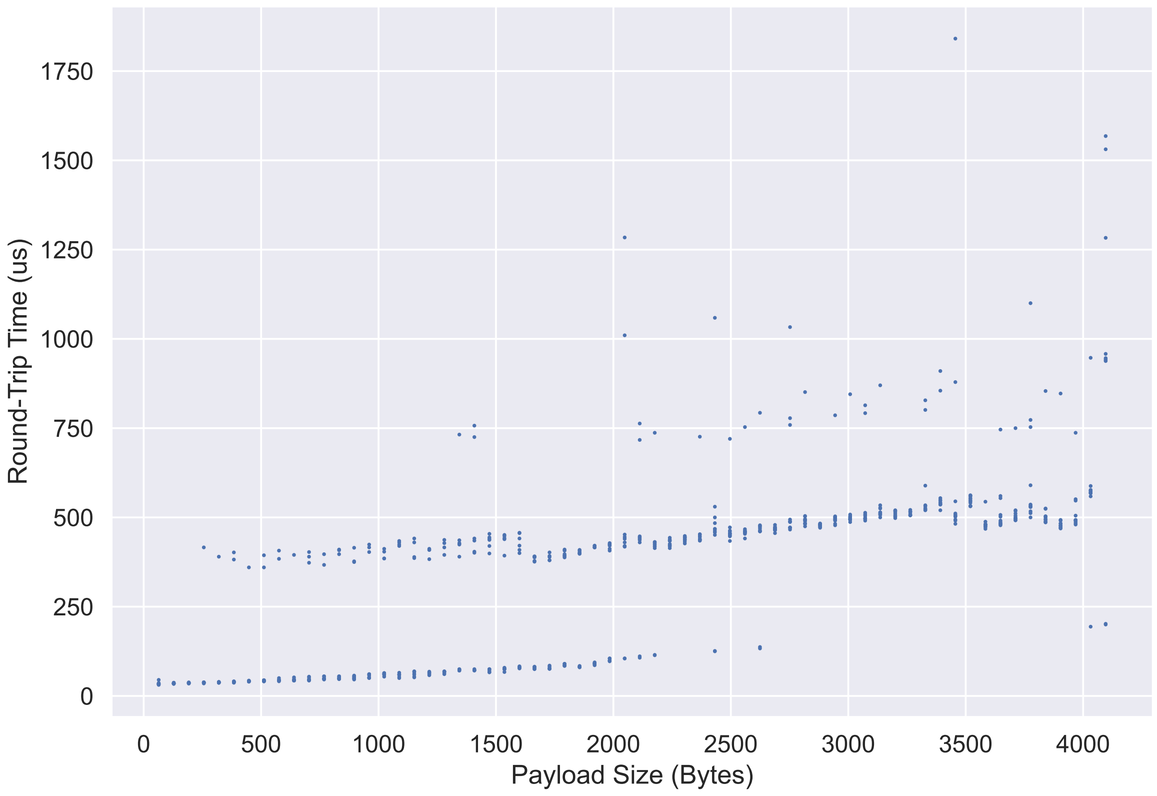

Figure 11.3: Round-trip time of a nameservice RPC with different payload sizes. No capability is passed. The server and the client run on different cores connected by UMP-based nameservice channel. One outlier: 3964us when the payload size is 4096, is omitted for readability.

Chapter 12

Networking

12.1 The enet driver

In order to build the network stack, we decided to extend the existing basic enet driver, as this seemed to be the best way to keep everything network-related in one place. This enet driver module is started by init and runs in its own separate domain. The network stack itself is realized by stacking function calls; each of them constructing (sending) or parsing (receiving) the header of a different layer of the stack. Besides the checks mentioned in the following description, we obviously also implemented checks to make sure that sizes coming from another domain or the network are valid (e.g., to avoid out of bound accesses).

12.1.1 Initialization

Device Frame

As the driver requires access to a device frame (to communicate with the device), the according capability is created by init and then passed to it at a well-known location (see the shell section of the book). Hence, the driver can mount the device frame after it's start and the pre-existing driver functions can take care of the communication with the device.

Device Queues

As explained in the book, buffers have to be added to/be removed from a device queue in order to send/receive packets. At the start of the driver, memory has to be allocated for and assigned to the device queues. There are 512 buffers for the sending queue and 512 buffers for the receiving queue. While a size of 1536 would be sufficient for them (i.e., maximum packet size), we decide to use buffers of size 2048 instead in order to facilitate header addition (see 12.1.2). After registration, the memory is also mapped into the user space of the driver (to allow access to the data stored in the buffers). As a precaution, we decided to map the receive buffers only using the read flags (i.e., without write). A similar thing is not possible for the write buffers, as they will also have to be read because checksum calculation might include payload data that has already been added to the buffer.

Managing the receive/send buffers

Note, that these buffers are intended to be reused. Hence, it is crucial for the driver to keep track of which buffers are currently not in the queue and hence, can be used during future operation.

For the **receive buffers** this is fairly simple, as they will stay in the queue almost all of the time. Only after a packet has been received into them, they can be dequeued, read (potentially storing data somewhere else for later use) and added back to the queue again to receive the next frame.

For the **send buffers**, things are more complicated as they will be outside of the queue most of the time (i.e., waiting to be filled with a packet and sent out). Hence, we use a stack to which all buffers are added initially. Whenever a send buffer is required, it can be poped off the stacked and be pushed back onto it after the sending has been completed.

Choosing an IP and MAC

Before we can communicate on the network, we choose an IP and MAC address. These values are variables in a struct and could be changed during runtime (but are not presently). Initially, the IP is set to 10.0.2.1 and the MAC is set to the MAC address that has been read from NIC. Note, that we will only consider packets that are sent to this precise MAC and IP (except for ARP, where we also listen on the broadcast MAC, see **??**). Especially, we do not support a subnet-mask and hence, broadcast IP packets will be ignored.

12.1.2 Sending through the different network layers

We will now give a brief overview over the different network layers. This will be done top-down (i.e., starting from UDP/ICMP) as this is the order in which the functions are called (and hence they expect parameters from the higher levels). The functions will pass a sending buffer through the different layers, which is augmented by the appropriate header at each layer. Hence, each layer will consider the buffer to already contain all the payload that is expected to end up in the packet which is constructed by it. Of course, also additional information that will be required by the lower layers will be passed along while no longer required information is not (e.g., the UDP function passed the IP but not the port to the IP function). Note, that all layers are non-blocking in the sense, that the unresponsiveness of another host will not hinder a call from returning (especially see 12.1.2).

Allocating a buffer

As a first step, a sending buffer has to be allocated. This is done by taking the top element of the previously mentioned stack. However, the very last element of the stack is reserved for ARP requests, as otherwise it might become impossible to send any more packages because many packets are stuck at IP level (see 12.1.2). This possible delay at the IP level also causes problems for enqueuing/freeing buffers again (as they can neither be enqueued yet nor be freed right after the call because they will be needed later on again). Note, that enqueuing a buffer into the device queue will make it available again later (see 12.1.5). To solve this issue, we implemented the following rule:

"A function call will take care of enqueuing/freeing the sending buffer (eventually) iff it succeeded."

This invariant is just passed down the stack. Either, the lower level function succeeds (and either enqueues the buffer now or makes sure it will be either enqueued/freed in the future) in which case also the higher level function will succeed, or it fails and the higher level function will fail with the same error code. Finally, at the lowest level (ethernet), the buffer is actually freed upon success.

Writing data

After allocation, the `valid_data` pointer will be set in a way that one can still write exactly the maximum supported packet size of 1536 bytes starting from it. This ensures, that we can send any packet we could send if we set it to the very start of the buffer and, at the same time, have as much space for potential headers as possible.

Now, the data can be copied into the buffer (as it will be the payload of the highest level function call), and the `valid_length` can be updated accordingly. This buffer is now all setup to be passed to the network stack functions.

UDP

This function can be used to send a datagram to another host. The UDP header looks as follows:

```
struct udp_hdr {
    uint16_t src;       // source port
    uint16_t dest;      // destination port
    uint16_t len;       // total length (including header)
    uint16_t chksum;    // checksum
};
```

In the function call, the header (with the passed values filled in) is added to the beginning of the buffer. Additionally, the checksum is calculated over this header, the payload and the following pseudo header:

```
struct udp_pseudo_hdr {
    ip_addr_t src;      // source ip
    ip_addr_t dst;      // destination ip
    uint8_t zeroes;     // 0000 0000
    uint8_t protocol;   // 17 for UDP
    uint16_t len;       // same as in udp_hdr
};
```

This ensures, that the UDP checksum also includes some additional fields usually only present in the IP header. However, this header will not be send along with the UDP packet (and is used only once for the checksum calculation). In UDP, the calculation of the checksum is optional (at least for IPv4), but we decided to implement it anyway. After filling the calculated checksum into the actual UDP header, the buffer is passed to the IP layer.

ICMP Echo

This function can be used to send a small ping request/reply to another host. If the send buffer contains a payload this will be used as a payload of the ping packet. The ping packet has the following header:

```
struct icmp_echo_hdr {
    uint8_t type;       // 0 for reply, 8 for request
    uint8_t code;       // code (0000 0000 for us)
    uint16_t chksum;    // checksum
    uint16_t id;        // id of the packet
    uint16_t seqno;     // number identifying a sequence of packets
};
```

The value for the type field has to be passed as a parameter (as we could send a ping request/response), the code is only used for other types and hence, set to 0 for our types. Our implementation takes id and seqno as a parameter, as we do not currently support sending out pings from user-space, we did not yet come up with a policy on how to set them for an echo request. In a response the values from the request are reused. After adding the header and calculating the checksum (over header and payload), the buffer is passed on to the IP layer.

IP

This function is responsible to encapsulate the passed payload into an IP packet and to send it via ethernet. The IP header looks as follows:

```
struct ip_hdr {
    uint8_t v_hl;       // version/header length
    uint8_t tos;        // type of service
    uint16_t len;       // total length (including header)
    uint16_t id;        // id of the packet
    uint16_t offset;    // stores information on fragmentation
    uint8_t ttl;        // time to live
    uint8_t proto;      // encapsulated protocol
    uint16_t chksum;    // checksum
    ip_addr_t src;      // source ip
    ip_addr_t dest;     // destination ip
};
```

As our implementation of the IP protocol is quite limited, many of the values are assigned fixed values. For now, only IPv4 is supported and additional options are not supported (which fixes the header length to five four byte words). Also, all packets will be sent out without additional `tos` information (i.e., the field is set to 0). Moreover, as fragmentation is not supported, the `offset` field will not be used (except for the `DF` ("Dont Fragment") flag which will be set at all times (to avoid having to pass it as an parameter for ping where it should be used)). `ttl` will be set to 255 (for better debugability); `proto` will be set depending on whether the function was called by UDP or ICMP. For the `id` of the packet we use a 16 bit counter in the driver state, which is increased after each packet that has been sent. The `src` is set to the ip currently stored in our driver state and the destination ip is passed as an argument. Lastly, the `len` is added and the checksum is calculated and filled in.

Unfortunately, we are not quite done yet as we only know the IP but not the MAC address of our destination. Hence, first a so called ARP query has to be performed in order to obtain the MAC address. As waiting for a response could take a long time (and possibly, none will ever arrive), we came up with a way to defer the actual sending of the packet until an answer has been received and integrated that into the ARP cache (which also is implemented in this function). Hence, this function does not always send out an ethernet packet instantly.

ARP cache

This cache is implemented using a hash table, mapping IP addresses to hardware addresses. Whenever an IP packet is to be sent out, first the cache is consulted. In case of a hit, the hardware address is used to send out the packet directly. Otherwise, a new (empty) entry is created, the packet is put into a linked list in that entry and a ARP query is sent out. If there already is an entry without a MAC address, the buffer will be added to the linked list and another ARP query will be sent out. These pending packages will later be sent out as soon as the answer to the query is received (see 12.1.3). It is important to see, that this can potentially result in congestion, if all sending buffers are pending inside the cache. This is the reason for the last sending buffer to be reserved for ARP (as an ARP request can always be sent and hence the buffer is guaranteed to become available again). Unfortunately, this design has two (related) drawbacks:

1. Packets for unreachable IP addresses will never be discarded
2. ARP requests are only sent once per packet

Both of these issues could be resolved by introducing time stamps into the entries and regularly traversing the hash table to resend ARP queries/remove too old entries. However, this has not been a problem during our practical tests and we hence decided against implementing this at the present time.

ARP

This function will be called by the ARP cache, whenever an actual query has to be performed as well as after receiving an ARP query (in order to answer). The header once again has many fields, but most of them will be set to constant values due to the limited implemented functionality:

```
struct arp_hdr {
    uint16_t hwtype;            // hardware type (1)
    uint16_t proto;             // protocol (0x0800)
    uint8_t hwlen;              // hardware length (6)
    uint8_t protolen;           // protocol length (4)
    uint16_t opcode;            // op code (1/2)
    struct eth_addr eth_src;    // source MAC
    uint32_t ip_src;            // source IP
    struct eth_addr eth_dst;    // destination MAC
    uint32_t ip_dst;            // destination IP
};
```

These constant values are the hardware type (1 for ethernet), the protocol (0x0800 for IP) and hardware/protocol address lengths (4/6). The `opcode` is passed as an argument and indicates whether it is an ARP request (1) or response (2). Source MAC and IP are set to the values stored in the driver state and the destination MAC and IP are passed as arguments as well (note, that the query MAC (i.e., all 0x00) is used for a request).

After the header has been added, the buffer is passed to the ethernet layer. In case of an ARP request, the broadcast MAC address (all 0xff) will be used.

Ethernet

This function performs the actual sending of an ethernet frame (by enqueueing to the hardware queue). Before that, a small header will be added:

```
struct eth_hdr {
    struct eth_addr dst;         // destination MAC
    struct eth_addr src;         // source MAC
    uint16_t type;               // type of following header
};
```

Here, the `src` will always be set to the MAC address currently set in the driver state, `dst` will be passed as an argument as well as `type` (either 0x0800 for IP or 0x0806 for ARP). Technically, also a 4 byte checksum would have to be added after the frame, but this functionality seems to be performed by the network card itself (as the frames are correctly received on the other end without explicit calculation of the checksum). Afterwards, the buffer is actually enqueued in the device queue and the function returns.

12.1.3 Handling through the different network layers

After this brief overview over the different layers from a sending perspective, we will now consider the passing of a receive buffer through the handling side of the stack. However, we will not repeat the header declarations and kindly refer the interested reader to the previous section. Also, this pass will be bottom-up, as this again is the order in which the buffer is passed from layer to layer. Each layer will remove and interpret the according header from the buffer and pass the required parameters into the higher functions. Unlike the sending functions, some handling functions might require a moment to complete (e.g., because a ARP request enabled many pending packets to be finally send out). Note, that the handlers will never free an receive buffer; this task is left to the caller after the call finished.

Obtaining a receive buffer

As a first step, we have to successfully dequeue a receive buffer from the device queue. This buffer can then be passed to the ethernet handler.

Ethernet

The ethernet header is read and the destination address is checked. If it does not match the current one in the driver state (or broadcast in case of ARP), the packet is dropped. The dequeued buffer still seems to contain a four byte trailing crc value, which will be ignored (but removed and not passed on to the next layer). Afterwards, the buffer is passed to either the ARP or IP handler or dropped passed on the `type` field.

ARP

The ARP header is read and it is checked, that the fields have the correct values for our specific implementation (the constant values that were described for the sending previously). Otherwise, the packet is dropped (as we could not extract any meaningful information out of it). If the buffer contained a request, we check whether we are the destination of that request (i.e., the ip matches) and if so, send out an ARP response. If the buffer contained a response, we check, whether the **ARP cache** is currently waiting for that value and, if so, add it to the cache entry and send out all packages that were pending for this IP (which might take a moment). In all other cases, the packet is dropped. Note, that this behaviour implies, that a value will never be updated once it has been stored in the cache. On the other hand, this provides more stability while developing and reduces the number of requests sent drastically.

IP

The IP header is read and verified. In particular, it is checked, that it is version 4, does not include any options, is not fragmented (but might have the Don't Fragment bit set) and has a valid checksum. If any of these conditions is violated, the packet will be dropped. The `tos`, `ttl` and `id` fields are ignored, as we do not support priority for packets, packet redirection or fragmentation. Next, it is verified, that the destination IP matches the one in the driver state. Otherwise, the packet will be dropped as there is (as mentioned before) no support for broadcasting addresses. Lastly, the `len` field is used to potentially shorten the packet. This can be become necessary due to the padding of frames at the ethernet level. Finally, the packet is either passed to the UDP/ICMP Echo handler or dropped based on the `proto` field. Both handlers will be passed the source ip address as a parameter.

ICMP Echo

This method will interpret and verify the ICMP header. This means, that the `code` is 0 and the `chksum` is valid. Then, depending on the `type` field it will either send an echo reply using the same `id`, `seqno` and payload or drop the packet. As there is currently no possibility to send ping messages from another domain, there is nothing useful that could be done with an echo reply instead (apart from printing it for the demo/debugging).

UDP

This handler will first verify the checksum (and therefore generate the pseudo header again) if there is one. It will then take care of passing the payload along with the source IP and port to the domain listening on the `dest` port (if there currently is one). Otherwise, the packet will be dropped.

12.1.4 Communication with other Domains

The communication with other domains is realized using the nameservice (see 11).

Passing incoming Data to another Domain

The data that arrived via UDP will be send via a nameservice channel. Therefore, the driver maintains a table mapping port numbers to nameservice channels (which are pointers). A `NULL` indicates, that the port is currently not listened on. Hence, the UDP handler can lookup this table, and if there is an entry, send the packet to the channel along with information on the source ip and port.

While this table might seem large at first sight, it is only 512 KiB (64k x 8 bytes per pointer). This is small compared to the 2 MiB spent on receive/send buffers.

In order to built up this table, the driver needs to be somehow messageable from other domains. Therefore, the it registers the name "enet". We defined a custom message struct (as well as one for the answer), which can be send to the service in order to perform network operations. There are several operations:

1. **Create** This message will try to create an entry in the aforementioned port table (i.e. start listening to a port). As an argument one can either pass a specific port or 0 for any port. Further, a name under which a response handler is reachable has to be passed. The driver will then establish a channel to this name, enter it into the table and return a confirmation with the actually assigned port. If the port was already registered by another domain (or not a single port is left) an error message is returned. In order to find an empty port for the case of an "any port" request, we maintain a pointer to the last found port (initialized to 40000) and start searching for an empty port from there. This basically employs a first-fit like search.
2. **Destroy** This message will remove an entry in the port table (i.e. set it to `NULL` again). Unfortunately, due to the lack of access control, a domain could steal a port from another one by first destroying the socket (because the driver has no way to verify the caller). A way to solve this issue would be to use capabilities. We have tried to do this, but quickly realized that it would cost too much time and hence, accepted the potential for misuse by a malicious domain.

3. **Send** This message can be used to send an UDP packet to a remote host. For the call, a socket (i.e., a port the program is reachable under) has to be provided (and potentially be allocated before). The reason to not condense these two calls into a single one is to keep the depth of the nested RPC calls low (and also sockets will be reused this way). Additionally, an endpoint (destination IP and port) and the data to be sent have to be passed.

Note, that these are only the calls that the driver understands. We provide a clean interface that abstracts over the use of the nameservice and provides the functionality of each of these messages from a single call (see) for more details.

12.1.5 Main loop

After the Initialization, the driver will begin executing the main loop. Due to time constraints, we did not implement support for interrupts and hence, the main loop will consequently poll three things:

1. The **receiving device queue** is polled and if there is a new packet available it will be passed to the ethernet handler.
2. The **sending device queue** is polled and if there is a new buffer available (which just finished being sent) it will be added to the buffer stack again. This is, why enqueuing a buffer is enough to make sure it will be freed again (because the device will eventually hand it back).
3. A call to the non blocking event dispatch is executed. This makes the driver responsive to incoming messages from the nameservice.

12.2 Accessing the Network from other Domains

In order to enable other domains to easily perform networking activity, we implemented a small library, that takes care of communication with the driver. It provides three functions that resemble the three messages mentioned before:

1. **Create** This method is passed a port (to be requested) and a listening function (which will be passed the senders ip, port and the actual data upon receiving a packet) and returns the socket that was actually allocated.
2. **Destroy** This method is passed a previously created socket and will take care of its destruction.
3. **Send** This method will send data to a specified endpoint using a previously allocated socket.

Avoiding deadlocks

All of these functions will use the same channel to the driver (which is only created upon the first call). The create method will use the pid together with a hard-coded string to temporarily register a name (which can be assumed to be unique) for the callback. Further, right before the RPC call, a background thread will be spawned to handle the channel creation that will be started by the server during the call (otherwise, a deadlock occurs). We also provide a wrapper receive handler for the nameservice that will call a function with a UDP specific signature after passing an incoming message in order to facilitate listening. Afterwards, the thread is informed to stop (via a shared boolean) and waited for to be exited. This way, it is ensured, that we only have background threads running when we really need them.

Sending Messages during the Invocation of the listener

In order to allow sending messages during the invocation of the UDP listener (e.g., for the echo server), the driver will also start such a background thread upon starting (because the domain might call another send while the RPC call is active). Note, that this one thread is sufficient at all times, as it is guaranteed that at most one domain is an UDP receive handler at any time. Unfortunately, the RPC call uses the default waitset, so the background thread might first handle some other events before dealing with the response. This could potentially slow down the driver (as it is blocked during RPC communication). The solution would be to create a separate waitset, but during our practical testing we did not encounter any issues due to this and hence, decided against this measure.

12.3 Two demo programs

We also provide two demo programs to test functionality:

echo_server

This is a basic UDP echo server that was also requested explicitly in the book. It takes a port as an argument and listens on it. As soon as a message is received, it will return the very same data.

nchat

This is a basic UDP chat programm that allows to have a conversation over UDP. It also takes a port as an argument and listens on it. Then, it waits for another host to connect and prints everything that is received. It also allows the user to enter data, which will be sent out to the last host that sent a message as soon as enter is pressed (or dropped if there is none). It can be stopped using CTRL + C.

12.4 Performance Evaluation

Ping latency

For the first plot (figure 12.1), we measured the time it takes to ping the board from the host (with no payload in the ping packet). This has been repeated a hundred times in one second intervals.

One can see, that all runs either took around 0.35 or 0.65 ms. This can be explained as follows: the RTT of our connection is in the region of 0.3 to 0.4 ms. Sometimes, an ARP request is performed first (by the host), resulting in approximately twice this time. Measurements with other computers over the same cable showed a similar RTT, so we conclude that this value is good.

UDP latency

Next, we measured the time it takes to send an UDP packet with no payload from our computer to the board and back (figure 12.2). This has been repeated a hundred times (without any gap in between) resulting in the following plot:

One can see two things: First, the RTT is a bit higher than for the pings. This can be explained by the fact, that the data will leave the driver and be transferred to another domain. As the RTT is still very low, we also believe this latency to be sufficient. Second, the values are concentrated in a single peak, with one outlier. This outlier is the first request, for which the ARP request had to be performed (as we did not have any gap in between packets, the host will not re-query this at all).

UDP throughput

In order to measure the UDP throughput, we sent 100 packets of size 1472 (the maximum size without requiring fragmentation) as well as an acknowlegment from the other side as soon as everything was received and measured the time passed on the host. For receiving packets on the board, we measure 47.1 ms and for sending from it 47.9 ms. Hence, the throughput is

$$8\frac{\text{bits}}{\text{byte}} \cdot \frac{1472\,\text{bytes} \cdot 1000}{0.0471\text{s}} \approx 25002123\frac{\text{bits}}{\text{s}} \approx 25.0\,\text{MBps}$$

for receiving and

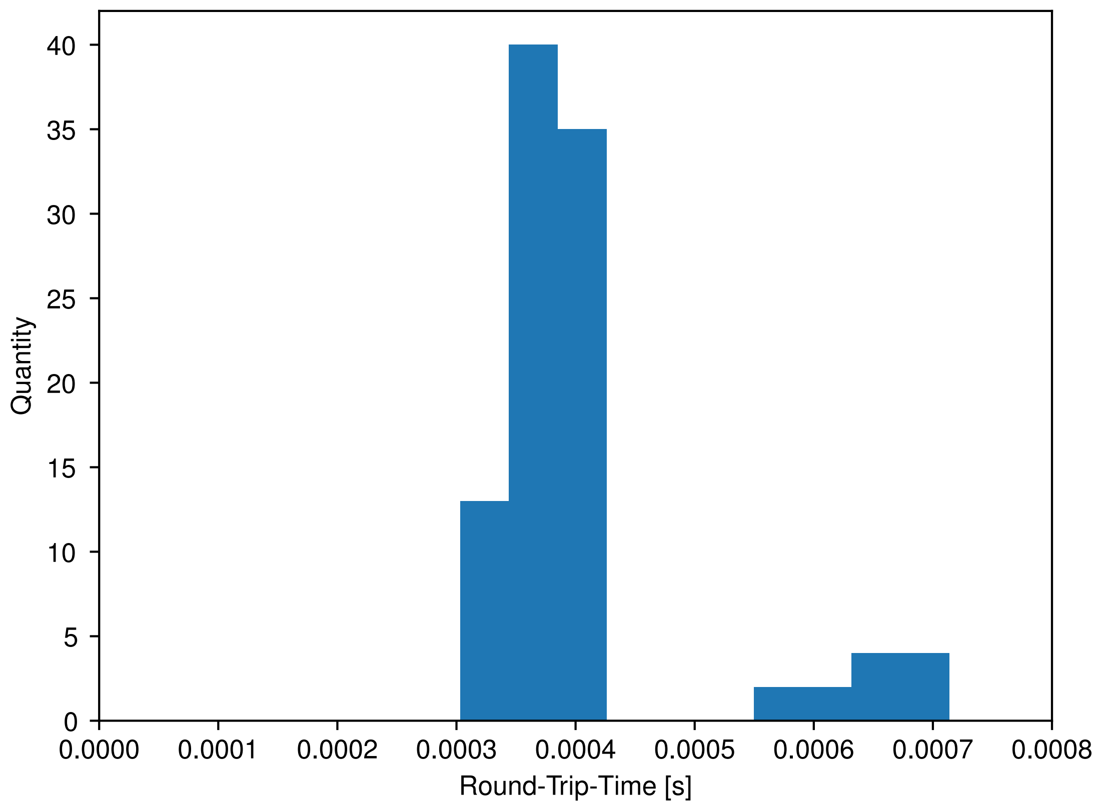

Figure 12.1: ICMP RTT from PC to board and back with no payload, 100 runs

$$8\frac{\text{bits}}{\text{byte}} \cdot \frac{1472\text{ bytes} \cdot 1000}{0.0471\text{s}} \approx 24584551\frac{\text{bits}}{\text{s}} \approx 24.6\text{ MBps}$$

for sending.

This limit to the throughput seems to have two causes: First, our driver is blocking while the handler processes the data. Hence, a context-switch is required for every single packet. We could increase performance here by buffering a few packages before actually calling the other side. Second, the throughput of the RPC would have to be increased further.

Known Issue

In the previous section, we only sent out a very small amount of data. The reason for this is, that the current implementation of the RPC calls is using malloc, which seems to reduce performance over time (and, more importantly: with the increasing number of bytes sent) (see 6.7.2). If we transmit larger numbers of packets, the throughput is reduced by a lot. Hence, we decided to only benchmark the fast performance in the beginning, as the problem in the ring buffer could be resolved (and performance is getting way worse fast).

12.5 Extra challenges

Due to the design of our shell it was not possible to implement remote UDP login. Also, our network stack lacks support for TCP due to time constraints.

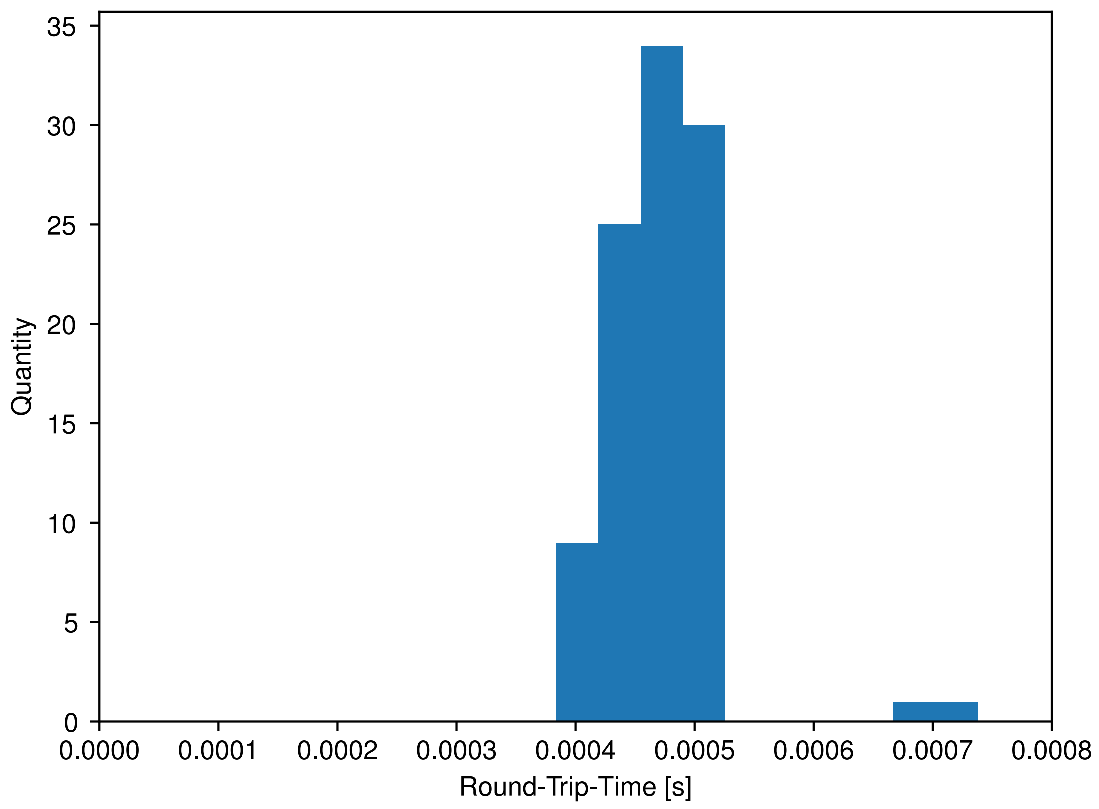

Figure 12.2: UDP RTT from PC to board and back with no payload, 100 runs

Appendix A

Appended Chapter

A.1 memtest

`memtest` is a small utility for testing integrity. Memtest also can make use of user level threads in order to stress test both the memory allocation/paging system as well as the user level threading library. The memtest program accepts a set of command line arguments that alter its behaviour:

```
Usage: memtest [<options...>]
Options:
    -background              Run in the background, returning to the shell
                             while memtest is running.
    -size=<decimal>          Set the size of the memory region in 4-byte
                             integers (as a decimal number)
    -hex-size=<hex>           Set the size of the memory region in 4-byte
                             integers (as a hexadecimal number)
    -pages=<decimal>         Set the size of the memory region in 4-kB pages
                             (as a decimal number)
    -threads=<decimal>       Set the amount of threads to start
    -data=<hex>[,<hex>...]   Set the data to write to the tested memory in
                             4-byte integers (comma separated list of hex values)
```

Every thread started will receive the specified amount of memory and fill it with the specified data before reading it again and checking its correctness. The different variants of size are mutually exclusive and the last specified option will take priority over the others. Default size, if unspecified is 64k 4-byte integers, or 256kB. Default number of threads is 1. Default data is a single value of `0xdeadbeef`. If not specified, memtest will not run in the background.

A.2 Shell

The shell supports reactions to certain keys on the keyboard in order to enable command history and command line editing. Namely the following:

```
  Home         Moves the cursor to the start of the line
  End          Moves the cursor to the end of the line
  Left arrow   Moves the cursor one character to the left
  Right arrow  Moves the cursor one character to the right
  Up arrow     Scrolls back through the command history
  Down arrow   Scrolls forward through the command history
  Backspace    Deletes the character before the cursor
  Delete       Deletes the character after the cursor
```

In addition, the shell supports the following builtin commands:

```
  echo <token...>          # prints arguments separated by spaces
  exit                     # exits the shell
  ps                       # list running processes
```

```
kill <pid...>               # kills the specified pids
oncore <core> <command>     # runs a command on a specific core
time <command>              # approximately times a command
ls [<directory>]            # lists the contents of specified directories
mkdir <directory>           # makes a new directory
cat <file...>               # prints the contents of specified files
cd [<directory>]            # changes current directory
pwd                         # print the current directory
rm <file...>                # removes specified files
rmdir <directory...>        # removes specified (empty!) directories
touch <file...>             # creates specified files
write <file> <token...>     # writes arguments line separated into file
append <file> <token...>    # appends arguments line separated to file
san <path>                  # sanitises specified path to valid absolute
```

Lastly, the shell supports multiple commands per command line:

```
<command> [& <command>...]
```

Only the last command will receive stdin, the rest will be spawned as background processes

A.3 nametime

`nametime` is a utility to benchmark the communication through nameservice provided channels. Running `nametime server` will start a server echoing a timed response back to a client started with no arguments. The client will then print the measurement and exit.

A.4 enumservice

`enumservice` is a utility to enumerate registered services in Ozone. It takes an optional argument as the prefix of the service name to enumerate.

A.5 dummyservice

`dummyservice` is a utility to register a service in the nameserver. The argument is the name of the service. The program keeps running. In order to run it in the background, `&` another command after it so that it does not get the stdin.

A.6 nameservicetest

`nameservicetest` is an extended version of the provided test program that stress tests all the functionalities of the nameservice. It starts by registering a service, and it spawns 8 clients on all the cores in round robin. All clients try to bind to the nameserver and look up the service, which requires robust concurrency handling. In our evaluation, it works correctly. Each client sends a message and a frame capability with a text written inside. The server receives the message and the frame, prints them, and replies a message as well as another frame. The client prints the reply message and the frame.

A.7 echo_server

This is a small utility providing a basic UDP echo server. It takes a port as an argument and listens on it. As soon as a message is received, it will return the very same data and print a debug statement on how much data it received and from where.

A.8 nchat

This is a basic UDP chat programm that allows to have a conversation over UDP. It also takes a port as an argument and listens on it. Then, it waits for another host to connect and prints everything that is received. It also allows the user to enter data, which will be sent out to the last host that sent a message as soon as enter is pressed (or dropped if there is none). It can be stopped using CTRL + C.

References

[1] D. P. Bovet and M. Cesati. *Understanding the linux kernel*. O'Reilly, 2006.

[2] The FreeBSD Project. *FreeBSD Library Functions Manual TREE(3)*, July 2020.

[3] Ios7 Lite 2. Single cloud svg vector. `https://www.svgrepo.com/svg/82735/single-cloud`.

[4] lmproulx. Nuage / cloud. `https://all-free-download.com/free-vector/download/nuage-cloud_116075.html`.

[5] memed.io. Laser eyes meme maker. `https://memed.io/laser-eyes-meme-maker`.

[6] A. Singhania and S. Gerber. Virtual memory in barrelfish. Technical report, June 2017.