CPSC 436A: Final Report

Arpit Kumar
University of British Columbia

Milo Piccini Noble
University of British Columbia

Seraj Aldin Abo Sabbah
University of British Columbia

Zehao Lu
University of British Columbia

Introduction

This document outlines the design of our implementation for milestones 1-6. We discuss the reasoning behind our design decisions and detail the issues we encountered. We would like to acknowledge all the help we received from the CPSC 436A team along the way – thank you!

Milestone 1

> "Simplicity is prerequisite for reliability."
> Edsger W. Dijkstra

The task for Milestone 1 was to implement the infrastructure for managing physical memory using Barrelfish's capability system. This entailed coming up with our own representation of physical memory and defining all memory operations in terms of this representation. This section explains the design of our representation.

A Model of Physical Memory

In class, we discussed several approaches to modeling physical memory, including bitmaps, red-black and AVL trees, and a linked list. We decided to use a (singly) linked list representation for one reason and one reason only: simplicity. Given that this was the first milestone, each one of us was relatively unfamiliar with the codebase, thus we wanted to focus on a *correct* solution rather than an *optimal* one. We discuss improvements on our design in a later section .

Each node in the linked list is tied to a single "free" RAM capability (these are passed on to the 'init' dispatcher from the CPU Driver). A node can be in one of two states: *free* or *allocated* and stores the size of the free/allocated block. Note that while we do enforce that every node holds unique capability references and that these capabilities point to disjoint address ranges, we do not maintain any particular ordering of the nodes.

Allocating Memory

To allocate memory, we traverse our linked list to find a memory region that satisfies the allocation requirements, and retype the free RAM capability into another capability of the type specified during the memory manager initialization. Notice that capabilities for allocated memory regions are descendants of the free RAM capabilities. We capture this parent-child relationship in each node by storing a parent capability reference. This reference is set to `NULL_CAP` if the node is free (i.e just points to the free RAM).

Allocation requests may have specific alignment requirements and varying sizes which often result in situations where the allocation starts at an offset from beginning of the free region. Moreover, allocation requests will likely be smaller than the size of the free region. To ensure we don't leak memory, we insert padding nodes to keep track of these fragments. Figure 1 illustrates this process.

As to our allocation strategy, we employ the *first-fit* policy. Again, we made this choice since it was simple and the resulting fragmentation isn't too bad for our purposes. We briefly explored a the more efficient *buddy* allocation strategy but we postponed its implementation due to time constraints.

We would also like to point out a subtle bug that we faced when setting up memory allocation. Initially, upon an allocation, we used the same `capref` in our node for that allocation as the one we returned to the calling process. This caused weird errors where our memory manager lost track of large chunks of memory for no apparent reason. The fix was to insert a call to `cap_copy` to ensure the capabilities used internally in our list are not affected by the capability operations performed by the `init` process.

Freeing Memory

Freeing previously allocated blocks of memory as a whole is essentially the inverse of allocating them and entails coalescing the padding/leftover nodes into a single node. Again, since we do not maintain any particular ordering of the nodes

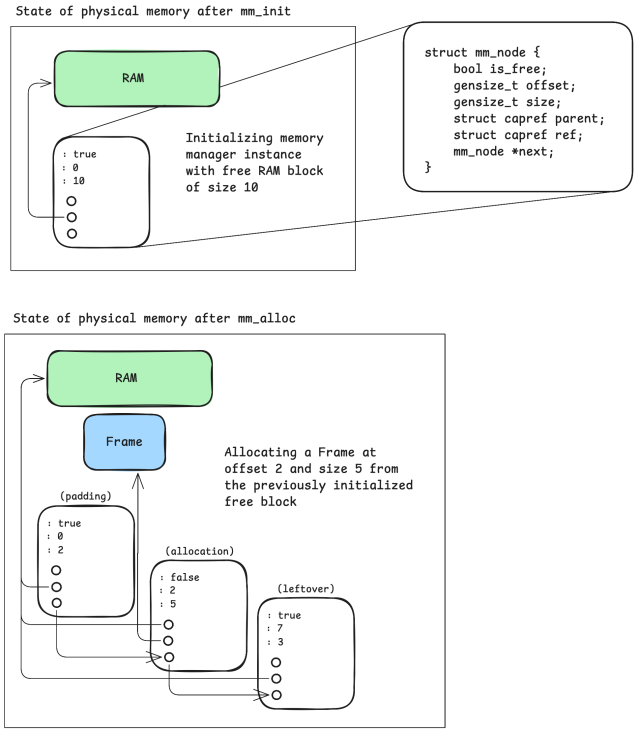

Figure 1: Keeping track of virtual memory regions

in our list, we are careful to not coalesce discontinuous memory regions.

Our implementation supports partial freeing of previously allocated blocks. To do so, we broke it down to following cases:

1. The partially freed address range *has the same start address* as its containing allocated range. In this case, we adjust the offset on the allocated node to align with the end address of the partially freed region. Note that we also call a retype on the allocated nodes's `ref` capability to make sure it is updated to to refer to this new, smaller region of memory. We then create a new capability for the free region and call `mm_add` to keep track of it.

2. The partially freed address range *has the same end address* as its containing allocated range. This case is identical to the previous one and is handled similarly – the only difference here is that the "second half" is freed. This is taken into account when retyping capabilities and updating the pointers.

3. The partially freed address range *does not have the same start or end address* as its containing allocated range. In this case, the allocated region is split into three. The first and third regions are free and are added back into the memory manager by calls to `mm_add` and the second region now refers to the smaller allocated region.

Notice that since we make calls to `mm_add` to keep track of the partially freed regions, once all sub-regions of an allocation are freed, we are unable to coalesce them. This is not ideal - we are essentially introducing "artificial" fragmentation. This may lead to a case where we are unable to satisfy an allocation request despite having enough memory since the representation of a free region is broken up into smaller pieces in our list.

Improvements

Here are a few improvements we can make to improve physical memory management:

- Decouple allocation policy implementation from the implementation of `mm_alloc` in order to experiment with different allocation strategies and determine which one gives the most favorable performance. We aren't quite sure how to judge the fragmentation that results from these policies apart from our theoretical understanding. For a more immediate improvement, we should use the *worst-fit* policy instead of the current *first-fit* since it achieves better fragmentation without any performance penalties.

- Memory allocation is a *very, very* frequent operation and a linked list isn't the most performant data structure. We are considering using an AVL tree structure indexed on the base addresses of a region or its size to keep track of the memory regions. This would greatly improve the time it takes to find a viable free memory region ($O(logn)$ versus $O(n)$).

Milestone 2

> "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."
>
> Brian Kernighan

The task for Milestone 2 was to implement a system for managing virtual memory. This can be broken down into the following modules

A Model of the Page Table

To map a given physical address to a virtual address on ARMv8, one must install an entry into the corresponding L3 page table. Due to the capability system present in Barrelfish, the user process wishing to perform this mapping must posses a capability to that particular page table. This means that the memory server must keep around an in-memory model of the actual page table.

There are a number of data structures one could use to encode this information, with the requirements being to maintain fast access and minimal memory footprint. There were two such structures that immediately stuck out:

- A **hash-table** would have a minimal memory footprint and provide very fast access. Deciding what to hash on here presents some difficulties, though. A simple approach would be to hash the index bits of the virtual address, and store the L3 page table along with a linked list of its parent page tables at that location. One could always find a necessary L2 page table by simply looking for the 0'th index L3 with those L2 index bits, and so on. This approach is fast! But still involves a number of layers of indirection. If there was a simple way to construct a *flat* hash table to keep track of all of the page table capabilities we needed, that would be an ideal approach.
- A **tree** in which each node contains an array of pointers to its children provides both a minimal memory footprint and constant-time lookup. There is a performance hit incurred from introducing between 3 and 6 levels of indirection (depending on whether the array of children is integral or allocated elsewhere), but the structure is simple and easy to manage. On top of the simplicity, the code for this structure models well the mental image of the tree of tables, making reasoning about it far easier. This is the approach we elected to follow through with. See Figure 2.

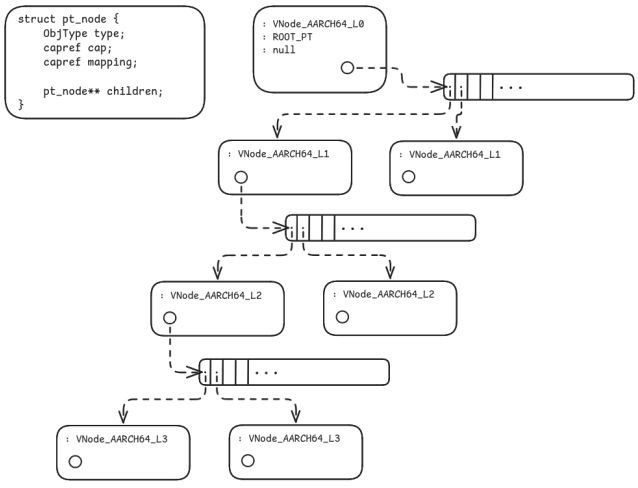

Figure 2: Our representation of the page table

Free, Allocated, and Mapped Memory

In order to manage and distribute virtual address ranges, we must have some structure to keep track of which regions of the virtual address space are free to be given out, and which are reserved. Additionally, since we require the ability to lazily back these regions with physical memory, we differentiate between allocated (reserved but not backed) and mapped (backed with physical memory). The operations that the virtual memory system must provide are as follows,

1. Find a free region of memory and allocate it.
2. Find a free region of memory and map it
3. Allocate a specific region of memory
4. Map a specific region of memory

Some of the operations turn out to be a lot more important than others in Barrelfish. Our key concern in designing this part of our virtual memory system was speed of backing virtual memory in the case of a page fault. Our design makes limited use of allocation and freeing of virtual memory regions, which is explained in the later section discussing our implementation of the **heap**. This means that the most frequent virtual memory operation that occurs is a traversal of our virtual memory region structure in order to determine whether an address that triggered a page fault should be mapped (is allocated) or should propagate the page fault as an error (is free).

We prioritize the speed of finding allocated but unmapped regions, and so an AVL tree storing ranges seemed like the obvious approach. By using a comparison function that checked to see if the provided range was a subset of any existing range in the tree, we could keep things balanced nicely. However, due to time constraints and messy development around this milestone, the initial approach we decided on was to keep a linked list of free and allocated regions to fetch from. A mapping operation, then, would remove a node from this list to be stored in an AVL tree. We failed to realize that this is an incredibly expensive operation to be performing for every page fault, and that optimizing the time it took to find a free region would make little difference to the performance of our allocator. See Figure 3 for a diagram of our region list.

The page fault handler itself receives upcalls from the barrelfish kernel anytime a given virtual address not backed by any physical memory is touched. The goal of the handler is to verify whether this page fault resulted from a valid memory access, and if so, to perform the mapping operation.

The Heap

Our heap functions by first reserving (allocating) a chunk of the virtual address space larger than the total available memory in the system. We then rely on the page fault handler and our other paging data structures to do the heavy lifting, lazily mapping chunks of this region when accesses occur. Using this approach, our actual `morecore` consisted of very

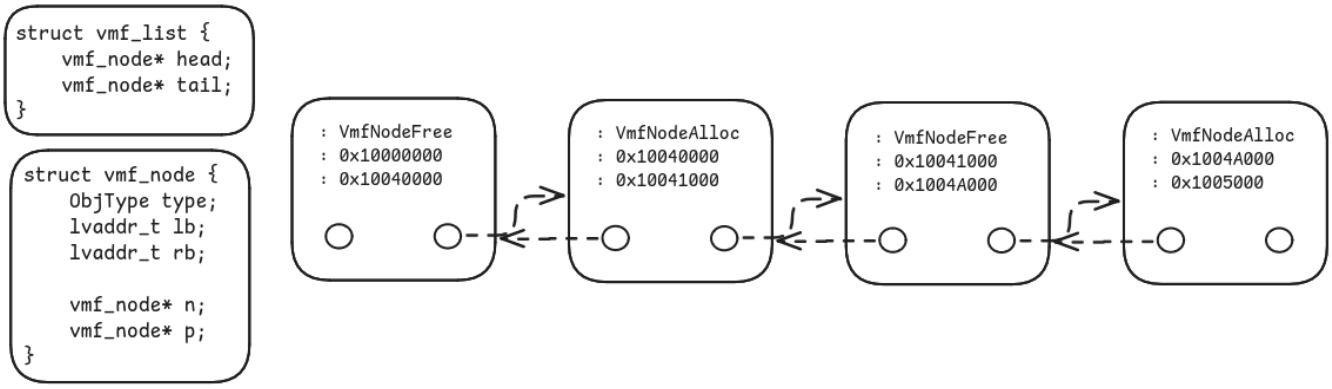

Figure 3: An initial approach: Keeping track of virtual memory regions

little code, and most of what was written serves setup and teardown.

Improvements

There are two lines of improvement we wanted to take with our allocation data structures. The first is simple, but halves the level of indirection involved in traversing our in-memory model of the page table. This is simply switching out the `pt_node** children` member of the `pt_node` struct for a `ptr_node* children[]` flexible array member. This allows us to maintain the small memory footprint of L3 page table nodes, which have no children, while simplifying the allocation code. We didn't end up implementing this change due to time constraints.

The second, more involved change to make to our allocation data structures was to rip out our `vmf_list` and simply use AVL trees to keep track of all memory regions. Our thought was that this would remove the $O(n)$ bottleneck on all of our memory operations and streamline the paging code significantly by unifying our logic for manipulating these regions. However, this did not end up being the case.

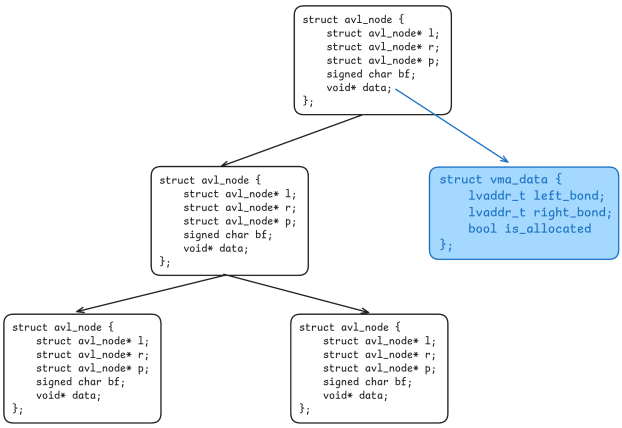

Figure 4: The free region tree

In testing, we found that using two separate AVL trees, one for free and allocated regions, the other storing mapped regions, was in fact slower in almost all cases. Allocation suffered a huge performance hit, as now instead of a simple linked list split, we had to rebalance an AVL tree. With the number of nodes we were working with in this memory allocator, this simply did not add up to a reasonable tradeoff.

The unexpected part of this is that mapping and unmapping remained largely unchanged. We can interpret from this that, even with the initial linked list implementation, the cost of AVL tree operations dominates. Figure 5 illustrates the performance characteristics of using a linked list vs an AVL tree to implement our free and allocated regions tracking data structure.

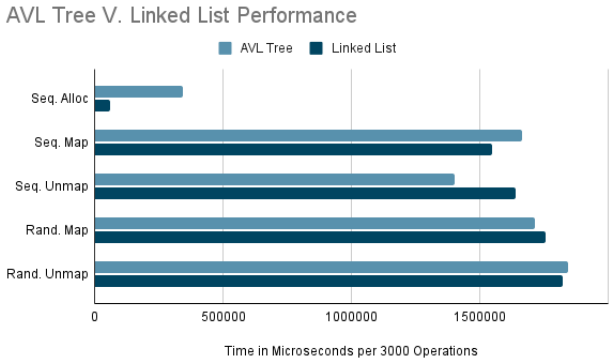

Figure 5: Performance characteristics

Milestone 3

> "Inside every large program, there is a small program trying to get out."
>
> Tony Hoare

Milestone 3 encompasses creating new processes and

managing their execution. Key tasks include setting up CSpace and VSpace, loading ELF binaries, configuring the dispatcher, and implementing process management features such as spawn, suspend, resume, and kill. The implementation covers all core functionalities required for Milestone 3, along with some additional process management tasks. The details are as follows:

Process Spawning

CSpace Initialization

- A two-level CSpace hierarchy is created for the child process.
- The Level 1 (L1) CNode is initialized using `cnode_create_l1`, and Level 2 (L2) CNodes are set up using `cnode_create_foreign_l2`.
- Key slots in the CSpace are populated with capabilities such as the dispatcher, root CNode, page tables, and argument frames, following the conventions specified in the Barrelfish documentation.

Listing 1: CSpace creation with L1 and L2 CNodes

```
errval_t setup_cspace(struct spawninfo *si)
    {
    // Create the L1 CNode
    errval_t err = cnode_create_l1(&si->
        l1_cnode_cap, &si->l1_cnode_ref);
    if (err_is_fail(err)) return err;

    // Create L2 CNodes and link them in the
         L1 CNode
    for (size_t i = 0; i < ROOTCN_SLOTS_USER
        ; ++i) {
        err = cnode_create_foreign_l2(si->
            l1_cnode_cap, i, &si->l2_cnodes[
            i]);
        if (err_is_fail(err)) return err;
    }
    return SYS_ERR_OK;
}
```

VSpace Initialization

- A virtual address space (VSpace) is set up for the child process.
- The L0 page table is created using `vnode_create`, and mappings for ELF sections are established using `paging_map_fixed_attr`.
- A default slot allocator is initialized to support subsequent memory allocation for the child.

Listing 2: Setting up the VSpace for the child process

```
errval_t setup_vspace(struct spawninfo *si)
    {
    // Initialize the slot allocator
    // ... removed code

    // Step 1: Allocate a slot for the L0
        page table
    //... removed code

    // Step 2: Create the L0 page table for
        AARCH64
    err = vnode_create(si->vspace,
        ObjType_VNode_AARCH64_l0);
    if (err_is_fail(err)) {
        // ... handle errors
    }

    // Step 3: Initialize paging state with
        the L0 page table
    err = paging_init_state_foreign(&si->
        process_paging_state, 0, si->vspace,
         &si->default_sa.a);
    if (err_is_fail(err)) {
        // ... handle errors
    }

    //... removed code that sets up RAM
        Capability

    return SYS_ERR_OK;
}
```

ELF Binary Loading

Loading the ELF binary into the child's virtual address space (VSpace) is a critical step in process spawning. The goal is to map program segments to specific virtual addresses as specified in the ELF file's program headers. This involves parsing the ELF file, allocating frames for the segments, and performing the necessary mappings. The steps include:

- **Parsing the Program Headers:** Each program segment is described in the ELF file's program headers. These headers specify the virtual addresses and sizes of the segments to be loaded into the process's VSpace.
- **Allocating Frames:** For each segment, a frame is allocated to hold its data. The size of the frame is determined by rounding up the segment's size to the nearest page boundary.
- **Mapping Frames to VSpace:** The allocated frame is mapped into the child's VSpace at the specified virtual address.

Challenges and Solutions

Handling VADDR_OFFSET Issues: During implementation, we encountered an issue where the virtual address specified in the ELF file was smaller than `VADDR_OFFSET`, which was previously assumed to be the smallest virtual address allowed for allocation. This assumption stemmed from the notion that the 1GB offset was reserved for booting. However, further investigation revealed that this reservation only applies to the `init` process, and the child process is free to allocate memory in the 01GB region. To accommodate this, we removed the relevant assertions from M2.

Unaligned Virtual Addresses: Another challenge was that the specified virtual address for some segments was not aligned to the page size (4KB). This violated the earlier M1 assertion that all allocated physical memory should be aligned to 4KB boundaries. To address this:

- The base address was adjusted using `ROUND_DOWN` to align it with the page size.
- The frame size was increased to include the unaligned portion by adding the offset difference.
- During mapping, the frame was mapped starting at the specified base address, leaving the initial unaligned portion unused.

This approach minimizes waste, as at most only 4KB of memory is unused.

Listing 3: Mapping ELF segments with alignment considerations

```
static errval_t elf_allocate(void *state,
    genvaddr_t base, size_t size, uint32_t
    flags, void **ret) {
    struct spawninfo *si = (struct spawninfo
        *)state;

    // Align base and size to page
        boundaries
    genvaddr_t aligned_base = ROUND_DOWN(
        base, BASE_PAGE_SIZE);
    size_t aligned_size = ROUND_UP(size + (
        base - aligned_base), BASE_PAGE_SIZE
        );

    // Allocate and map the memory
    // ... removed code

    // Map into the parent's VSpace
    // ... removed code

    // Map into the child's VSpace
    // ... removed code

    *ret = local_addr + (base - aligned_base
        );
    return SYS_ERR_OK;
}
```

Global Offset Table (GOT): Once the ELF segments are mapped, the Global Offset Table (GOT) is located using ELF section headers. The GOT base address is then used to initialize the dispatcher registers, enabling runtime resolution of global variables and facilitating shared library support.

By addressing these challenges, we ensured that the ELF loader correctly handles real-world binaries, even with complex memory layout requirements.

Dispatcher Setup

- A dispatcher control block (DCB) is allocated and mapped into both the parents and childs address spaces.
- The dispatcher is initialized in disabled mode, with fields such as core ID, domain ID, and the program counter set appropriately.
- The `armv8_set_registers` function initializes the GOT base and entry point for the dispatcher.

Listing 4: Initializing the dispatcher fields

```
// Access the dispatcher structure fields
struct dispatcher_shared_generic *disp =
    get_dispatcher_shared_generic(handle);
struct dispatcher_generic *disp_gen =
    get_dispatcher_generic(handle);

// Initialize dispatcher fields
disp->disabled = 1;
    // Start in disabled mode
disp_init_disabled(handle);
    // Initialize disabled state
disp_gen->core_id = disp_get_core_id();
    // Set the core ID for the child
disp_gen->domain_id = domain_id;
    // Assign the domain ID
si->pid = domain_id;
    // Store the domain ID in the spawninfo
    struct
```

Argument Passing

- Command-line arguments are parsed and stored in a dedicated arguments frame in the childs address space.
- The `spawn_domain_params` structure is populated with pointers to the arguments in the childs VSpace, ensuring proper alignment and null-termination.

Process Management

Start, Suspend, Resume, and Kill

- Processes are started by invoking their dispatcher capability, transitioning them to the `RUNNING` state. Every new process gets assigned a PID by the process manager.
- Suspend and resume functionality is implemented by manipulating the dispatchers state in the run queue.
- Processes can be terminated (killed) by removing their dispatcher from the run queue and setting their state to `KILLED`.

Tracking and Cleanup

- A linked-list data structure is used to keep track of processes, storing information such as name, PID, and capabilities.
- Although full cleanup of resources is not yet implemented, the framework is in place to traverse and reclaim allocated resources.

Optimization and Improvements

- Implement full cleanup of process resources and automatic updates to the process tracking structure upon termination.
- Improve the PID assignment as it may overwrite the running PIDs if the number gets too large

Milestone 4

> "The single biggest problem in communication is the illusion that it has taken place"
>
> George Bernard Shaw

Now that we have more than a single protection domain, we need a way to communicate between them. In a traditional operating system, one can imagine wanting to be able to request that another process be killed so that a task manager might work, for example. This inter-process communication is even more fundamental to the operation of Barrelfish due to the lack of services in the kernel. When we spawn a new process, if that process wants RAM, it must ask for it from some other process that has capabilities to RAM already. In some sense, IPC is the glue that holds Barrelfish together.

What We're Working With

Barrelfish comes with a library for transferring data and capabilities between dispatchers (that this is between dispatchers is important) already, in the form of Lightweight Message Passing. This library allows for non-blocking IPC by invoking a (retyped) capability to the other processes dispatcher, which pulls the kernel into the fray to work a little magic. On top of these raw endpoint capabilities, we have provided for us a channel abstraction, giving rise to "LMP Bindings" and the ability for a thread to wait for a message on such a channel.

Our goal with this milestone was to implement a fast and reliable framework for Remote Procedure Calls between protection domains, built on top of the LMP Channel abstraction. RPC aligns well with the kind of functionality we want out of this system: If a process wants more memory, we're fine with a synchronous call to request it. We also only need very small messages, since we can provide stubs for each of the functions. This means any single call can be made quite quickly. Given we'll be implementing printing to the terminal via this mechanism, our design has to minimize latency.

These operations are core to the function of Barrelfish, so we wanted them to be painless to use, easy to extend, fast, and most of all to seamlessly integrate with UMP (M6) when messages are passed across cores. We succeeded in some of these things.

A Messaging Layer Protocol

To define a protocol, we need a conception of what a message looks like. The underlying LMP endpoints support transferring up to 8 words at a time, and channels give us a way of blocking and waiting on the dispatch/receipt of these packets. To enforce structure on two ends of a connection, one needs data that describes structure. If there's just one type of message, this can simply be globally agreed upon and fixed. Given the number of possible RPC subroutines, we need some extra information to be sent with the message itself. We decided to sacrifice 1 word of payload space to pack in information about the length of the payload, its type (i.e. what sort of call we are making – this lets us agree upon structure on each end), and some other information that is needed for handling messages larger than a single payload. With a message format in place, the task of both the client and server becomes marshalling and unmarshalling messages, and making their respective local function calls – we have (semi) successfully decoupled our RPC from the underlying transport protocol.

Listing 5: RPC Structs

```
union rpc_hdr {
    struct {
        uint32_t pld_len;
        uint8_t type
```

```
        ...
    };
    uint64_t hdr_val;
};

struct rpc_msg {
    struct capref cap;
    union rpc_hdr hdr;
    uintptr_t pld[];
};
```

The Handshake

With an idea of *what* we want to send in mind, we have to actually establish a connection. To keep the latency of each message low, Barrelfish borrows the concept of an explicit 'binding' from LRPC. That is, an LMP channel is persistent over the whole lifetime of a spawned process. Producing this binding is not a particularly interesting design choice, as there's a relatively fixed order. However, as a consequence of our choice to use shared memory to handle large messages, we introduce a small amount of complexity. Since all OS services live in the `init` process (again, this is a design decision we made in light of time constraints), any spawned process must first share its endpoint capability to `init` which in turn returns the capability to the shared memory buffer used to handle arbitrary sized messages. This exchange of capabilities is what constitutes our LMP handshake.

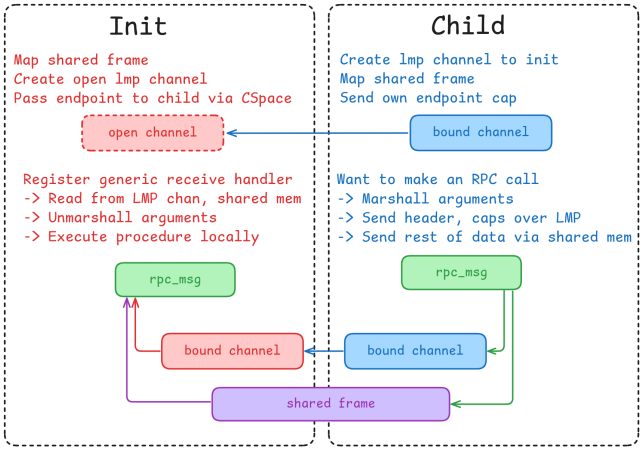

Figure 6: The LMP Handshake

The Big Picture

The final design of our message passing system can be seen in Figure 9. The `burb` library sits at the core of this, handling marshalling/unmarshalling, dispatch logic and receiving messages. `burb` is given an RPC message struct and returns an RPC message struct, hiding all of the underlying LMP logic away. Recall that our architecture stuffs all of the functionality of a memory server, terminal server, and process server into `init`. On the server side then, `burb` takes the form of a generic receive handler that is always registered, forwarding payloads to server-specific functions.

One key observation we made was that a given LMP channel is dispatcher to dispatcher, and hence the introduction of multiple threads on each process meant working with care. The simplest approach in this case was to enforce an invariant and make RPC send/receive pairs atomic. We have a mutex associated with each binding, and the thread making a call acquires the lock before sending a message, releasing it only after it has received a response.

Flaws

There are a number of flaws in this approach. The first and most obvious thing is that it severely limits the kinds of RPC calls we can make. Our invariant is that a call is always initiated by a client, and that this call may be treated as a single operation. This assumption was quickly violated when we considered implementing `proc_mgmt_register_wait`, an RPC call that tells a thread to suspend until a specified process is killed. This call is *open*, and with our current design, this could potentially intercept all other calls to the process hosting the suspended thread. We ended up not solving this issue. Perhaps more troubling was our assumption that the server would never want to send a message to a client without the client asking first. This immediately ruled out client-to-client communication via the server process and made a large chunk of M6 totally unviable.

It's worth noting that LMP channels are reliable, so we don't worry about messages failing to send. However, we have no time-out mechanism. If for whatever reason a thread is not able receive a message it expects, not only will it hang, but no further RPC calls will be able to be made by any thread on that process.

Lastly, as mentioned earlier, we decided to house all OS services in the `init` process. This is less than ideal because this limits concurrent access to distinct services. It also means that if the `init` process stops running, access to all services is lost. An ideal approach would have a process for each service along with a name server to enable service discovery.

Handling Large Messages

So what if we want to send messages larger than 7 words? There are a number of potential approaches. One choice is to simply split the message into many LMP messages, associate with each on a sequence number, and reconstruct the message on the other side of the channel. This is slow because we have to context switch for each message, and introduces plenty of opportunities for mistakes to be made in reconstructing the message. Another, arguably simpler ap-

proach, is to just share some memory between each process. In our design, we keep these messages synchronous and use LMP to transfer a header describing the message we've written to the shared memory frame. We support arbitrarily sized messages, but with an entire page of memory to work with, the vast majority of messages will only require a single read from this buffer. To illustrate the difference in performance, we have provided benchmarks in Figure 7, comparing shared memory to multiple LMP messages. This is cropped for readability.

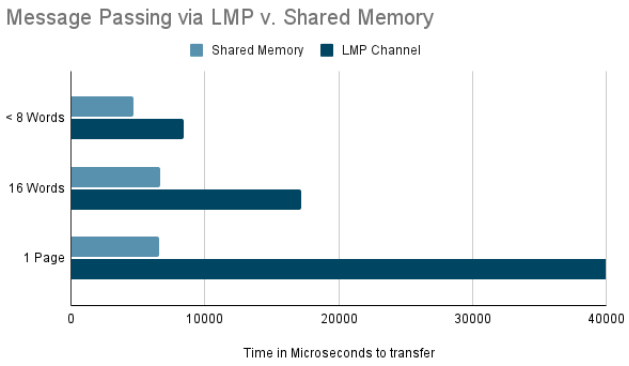

Figure 7: Performance characteristics

And now without cropping (clearly, the number of context switches in the approach using raw LMP payloads dominate the time it takes to share 1 page worth of data and is multiple order of magnitudes slower than using a shared memory buffer).

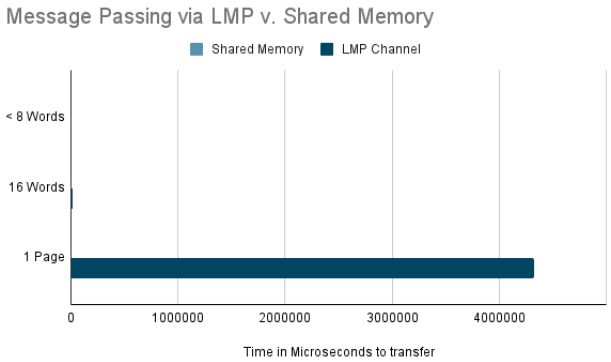

Figure 8: Performance characteristics

Milestone 5

> "The free lunch is over. Now welcome to the hardware jungle."
>
> Herb Sutter

The main focus of M5 was to bring up another core. This is where we start to get a sense for Barrelfish as a multikernel operating system. Barrelfish operates under the assumption that no operating system state is shared between cores. Each core runs its own CPU driver, each of which is like its own microkernel. This is a key part of why Barrelfish supports running on heterogenous hardware, as each CPU driver may be built for a different architecture. On each core, a privileged user-space process called the Monitor (in our case `init`) serves both to handle long-running operations that the kernel itself might not be able to handle (due to its stateless, non interruptable nature, we don't want to simply hang while a capability is deleted) and to communicate with monitor processes on other cores. With this architecture in mind, we also need a way of providing basic communication between cores. We will, in fact, use a shared memory frame (as in our implementation of M4, and as we will expand on in M6).

The Process

Much like M3, the approach here is fairly formulaic. There are a set number of steps one must take to boot a new CPU driver on a second core.

1. Allocate memory: We need to provide RAM for the kernel .bss segment, RAM for the init process on the new core, RAM for our communication channels, and RAM to retype into a Kernel Control Block (KCB).
2. Create KCB & Coredata data structures: The KCB is the object holding references to all of a running kernels state, while Coredata provides the boot parameters to the new CPU driver.
3. Load the Boot driver and CPU driver: Since there is no shared OS state, we need to load a whole new copy of the CPU driver into RAM. Further, the CPU driver assumes that there is a default kernel page table available upon boot, and that the MMU is enabled. This is handled by the boot driver, which does the last required bootstrapping required to jump into the CPU Driver.
4. Clean the cache: Since the boot driver starts with the MMU uninitialized, it's important to make sure that everything it needs access to is actually in RAM, not hidden away in some dirty cache line of our current core.
5. Call spawn: Here we invoke the kernel capability, which abstracts away all of the hardware magic, performing some boot protocol based on platform to start the boot driver. This then spins up the CPU Driver, and we have a new core running.

Communication

The communication channel between cores is a region of shared memory for which the details are passed via the

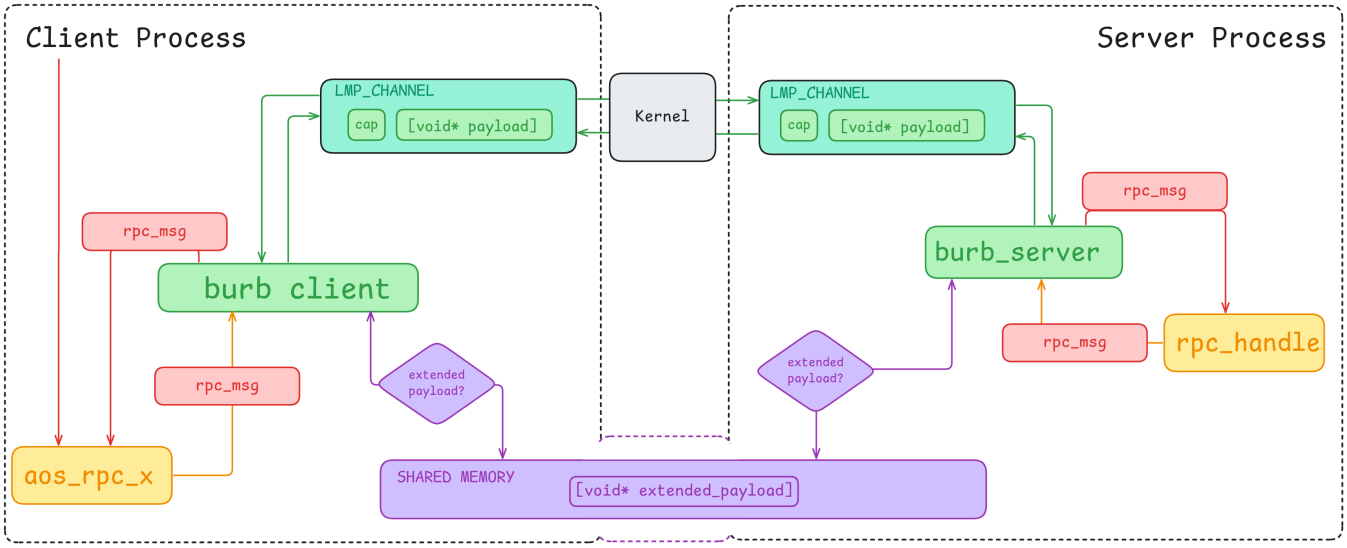

Figure 9: Our RPC System Design

CSpace when booting the new core. In this milestone, this channel was unidirectonal, meaning processes running on the BSP core may make requests to processes on other cores, but not the other way around. We realized in M6 that a simple way to implement bidirectional communication was just two have two producer-consumer structures, "facing" opposite directions occupying the same shared frame. The upper half could be dedicated to reading on core0, writing on core1, and vice versa.

To actually use this channel, and in the absence of any mechanism similar to waitsets, we decided it would be best to have the reader continuously poll the shared memory. Here we had two choices, either

1. Bring up a thread in each init process, or
2. Spawn a new process on each core to serve as a UMP server

Complications

Our initial approach was to work with a polling thread, since that fit our current design more cleanly (given that every server so far had lived inside init), but we ran into many issues with threading. As it turns out, we did *not* have working threads as we had assumed, and made a few fundamental mistakes in understanding. One of these was the misinterpretation of `_onthread` functions, not realizing that these were named as such not because they ran on each new thread, but that they were designed to run in a separate thread on the process they were invoked on. Our thread stack had also been allocated half the size it needed. We fixed these only upon tackling M6.

Since at this point threads were not working, we decided on the approach of spawning a new process. This meant that all messages over the channel had to be forwarded via LMP to init to actually be executed, giving up any performance gains that shared memory might have offered. This also led to poking a lot of holes in our M4 API, further complicating M6 work.

Taking "Turns"

Once we had a polling thread, we needed some sort of mechanism to make sure we weren't reading partially written data. For this, we borrowed an idea from Sequence Locks, and implemented a "turn" based synchronization system. This was a simple way of ensuring that a single-producer single-consumer queue is read on the correct order. We would learn in M6 that in fact memory barriers are key to synchronization across cores on ARM due to its weak memory consistency model. The turn is just a single bit in our buffer data structure that signals whether it is time to read or write. This "turn" based mechanism is demonstrated in Figure 10.

Further Challenges

Since we had no waitsets to rely on, our implementation diverged quite heavily from the event driven nature of our previous code. The ability to register a callback made reasoning about LMP communication quite simple, but in our M5 implementation we had to manually handle each message at time of receipt. This didn't fit well with the RPC interface and caused more holes to be punched.

Similarly, to get our unidirectional channel to function we had made many assumptions about communication. These compounded on those we'd made in M4, coming back to bite us in the implementation of M6.

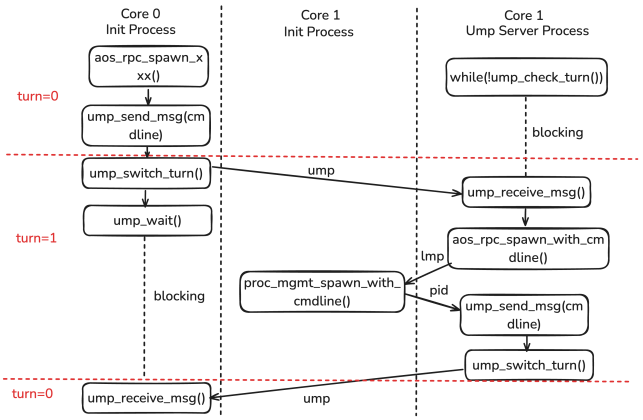

Figure 10: Call graph outlining the steps that take place when a request to spawn a process is made across cores

Milestone 6

> "I have not failed. I've just found 10,000 ways that won't work."
>
> Thomas A. Edison

Ah, the last milestone. Now that we have the ability to spawn an additional core, the goal of this milestone was to implement a full inter-core messaging and RPC system by extending our work from M4. We were fairly confident in our conceptual understanding of URPC, but faced several technical and non-technical challenges and we eventually ran out of time before we could implement a working solution. We try to describe our idea, what worked, and what didn't work in this final chapter.

The Idea

The idea behind this milestone was to take advantage of shared memory and cache coherency to implement a really fast messaging system between cores. While sharing memory is great when it comes to performance, we had to be extra careful in how we were handling reading/writing to shared memory given Barrelfish's weak memory consistency model. In particular, this meant our ring-buffer implementation had to make use of memory barriers to ensure that messages were written and read in the correct order. Figure 11 summarizes at a high-level how we envisoned M6 to work. A newly booted core receives a capability to a frame of with a size of two pages. One of these pages serves as the ring-buffer that this core will use to send messages, and the other half of the frame is used to receive messages. We will refer to these as the *producer* page and the *consumer* page going forward. Also, notice that arbitrary processes in each core cannot access the UMP prdoucer buffers directly. Instead, they must first send a LMP message to the `init` process, which would then serve as the "messager" between that process and the other core. Keep this in mind since this was a major pain point for us that we discuss later.

Since there was no LMP-like event-driven abstraction for UMP messages, we decided to spawn a thread in the `init` process for each core dedicated to polling the consumer ring-buffer for messages and handling them appropriately. This is where we uncovered critical, blocking issues in our work from the previous milestones...

Hurdles

We realized that we were unable to spawn threads! Any attempt at using Barrelfish's threading library immediately resulted in kernel panics and ugly red error messages polluting our logs. We will save you the details but the issue was that we never tested for threading in the earlier milestones and did not consider thread safety of our systems until towards the end of the course. Moreover, we (wrongly) believed that paging didn't need to be thread-safe since processes would only access the paging interface via `malloc` and friends - which are inherently thread-safe. We failed to consider the fact that the page fault handler and the process spawning methods heavily rely on the raw paging interface. Therefore, paging did, in fact, need to be thread-safe. We went ahead and made the slab/slot allocators thread-safe as well, just to be safe.

Just to go back to the point on testing – we believe we didn't write nearly enough tests across milestones, even though we knew better and always kept mentioning "we should write more tests". We attribute this to poor communication and distribution of tasks and the fact that we fell a little behind on the later milestones.

Initially, we tried to work around our threading issues by spawning a new UMP monitoring process on each core. This could have worked, but this meant that we would have to send an additional LMP message between this process and `init` – we felt this defeated the purpose of having fast shared-memory-based messaging since we would end up incurring an unnecessary context-switch cost.

Once we finally got threading working, we faced another issue with our M4 implementation – we never considered the case where the `init` process might want to push a message to a child process. Therefore, we were only able to achieve inter-core communication between the `init` processes of two cores.

What Worked

Clearly, not a lot was going our way in this milestone, but not all was lost! Our RPC message handlers from M4 were fairly decoupled from the underlying transport protocols and not a lot needed to be rewritten to implement the URPC message handlers. Moreover, we were able to use the idea of sharing memory to make M4 faster when it came to sharing arbitrary

sized messages. We are also proud of the effort we put in in trying to get M6 to work – a lot of sleep was sacrificed! But more importantly, we had fun and enjoyed every minute of time spent on this course.

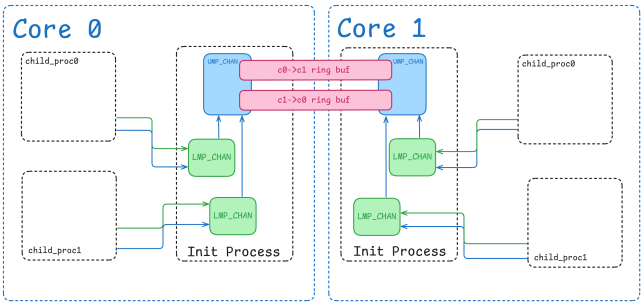

Figure 11: A rough outline of our (intended) URPC system

References

[1] Xieqing (2024) *AVL Tree Library*, https://github.com/xieqing/avl-tree, Accessed: 2024-11-15.