CPSC 436A: Finding Barrel Report

Bob Pham
University of British Columbia

Cathy Liu
University of British Columbia

Jiayin Kralik
University of British Columbia

Shibo Ai
University of British Columbia

1 Introduction

The *Finding-Barrel* project is a group-based assignment focused on designing and implementing a custom operating system based on the Barrelfish OS from ETH Zurich. This report documents our system's development process, detailing our design choices, implementation, testing strategies, and performance benchmarks across the core components of our operating system.

Our team followed a flat organizational structure, allowing each member to contribute broadly while enabling specialization for specific milestones. For each project milestone, one team member typically led by proposing the initial architecture and design. This proposal was then reviewed and refined collaboratively, ensuring a well-rounded approach. Once a design was finalized, tasks were divided among team members, with some focusing on core implementation and others supporting with testing and early planning for subsequent milestones.

The following sections discuss our design strategy for each component, the rationale behind our choices.

2 Overall Design

The overall design of the project is based on Barrelfish OS, developed at ETH Zurich. Barrelfish is a microkernel operating system with key features such as capabilities system, remote procedure call, etc., and emphasizes modularity and simplicity by pushing most services into user space, leaving the kernel clean and relatively lightweight. This separation of responsibilities ensures better isolation, making the system easier to maintain and extend.

In line with Barrelfish's philosophy, the design of key services in our project, including the management of physical memory, virtual memory, and processes, the spawning of processes and cores, and sending messages between processes and cores, prioritizes simplicity, robustness, and efficiency. We have deliberately chosen straightforward yet highly manageable data structures, such as linked lists and trees, to represent and manage critical system components. These data structures allow us to balance ease of implementation with the flexibility needed to address complex system requirements.

3 Milestone 1

In this section, we discuss the implementation of a memory and capability management system for our operating system project. Our design focused on simplicity and efficiency, aiming to facilitate secure access control across system resources.

3.1 Data Structure

The core data structure that supports our memory manager is a singly linked list. This linked list is used to represent the whole of the available memory in our system. Each node in the linked list carries the following meta-data:

1. The capability for the memory region that this node represents
2. A flag for whether or not a node is free or allocated
3. The base address for the memory region that this node represents
4. The size for the memory region that this node represents
5. The parent capability for the **original** memory region that the node was split from
6. The base address for the **original** memory region that the node was split from

Further, the linked list created and managed such that the nodes are arranged in ascending order by base address.

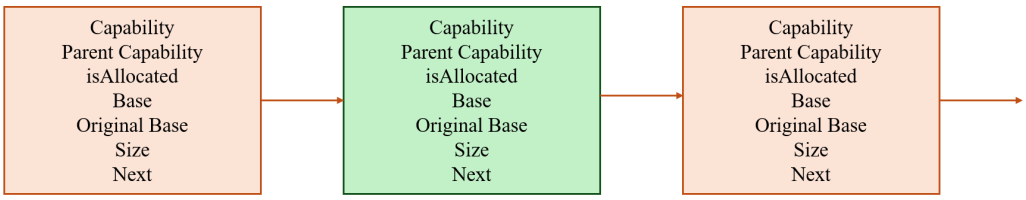

Figure 1: Linked List Memory Manager

We chose a linked list data structure to represent the physical address space, as it provides a clear and straightforward way to model the generally contiguous and sequential nature of physical addresses. Specifically, we opted for a **singly** linked list for its simplicity, as it requires less overhead in pointer management compared to a doubly linked list. Reflecting on this choice, a doubly linked list might have been advantageous in some scenarios, as it would allow us to avoid managing a "prev" pointer manually during traversal. However, both singly and doubly linked lists have a complexity for $O(n)$ traversal, so we concluded that the difference in performance would be minimal.

Road Not Taken

Initially, we considered a tree-like structure in which the parent node represented the memory region specified by the original capability provided during `mm_add`. When memory needed to be allocated through splitting and retyping, a child node would be created for the respective parent node, recording details like base address, size, and other relevant attributes.

However, we soon realized that maintaining a large parent node after splitting was unnecessary. For instance, the total size represented by the original parent node became less useful once memory was split, as knowing the remaining available bytes was sufficient. This tree structure introduced redundant information and added complexity without providing significant benefits. As a result, we opted for a simpler and more efficient design: a linked list.

3.2 Algorithms

This is a high-level overview of the different algorithms that are used during the allocation and deallocation of memory in the memory manager.

3.2.1 MM Add

During start-up, memory resources are given to the `init` process. The `mm_add` function is responsible for adding new memory resources, represented by a capability, to a memory manager instance. A metadata node (`mmnode`) is allocated to represent the new memory region, initialized with the capability's base address, size, and other relevant information, and marked as free. The function also checks for overlaps with existing regions in the memory manager, preventing duplicate entries for the same physical memory. If no overlap is found, the new node is inserted into the linked list in ascending order by base address to maintain an ordered structure.

3.2.2 Alloc/Alloc Aligned

Finding Available Addresses

The allocation of a slot is done using a first-fit approach. When finding an available slot, we traverse the linked list until we find a node that is marked as unallocated. We then do the following checks:

1. Check if it is large enough to support the size of the capability that we are trying to manage
2. If there needs to be special alignment (default to 4096), see if the size of the gap is large enough to account for alignment padding
3. If the allocation is for a specific range, check if the node is within the range
4. Check if the memory region overlaps with any existing allocations

Once the memory region is found, the node is split into up to 3 parts as needed, where the segments could be: the segment before the found memory spot, the aligned memory spot, and any remaining memory after the aligned region. These nodes are kept in ascending order of base address. Once this is done, the capabilities for the aligned and requested memory spot are retyped to match the aligned memory region found.

We chose the first-fit algorithm primarily for its simplicity, as it is straightforward to implement and manage. However, we recognize that first-fit is more likely to fragment memory at lower addresses in long term, which can lead to performance degradation over time. Alternative algorithms, such as worst-fit or next-fit, could offer improved performance in the

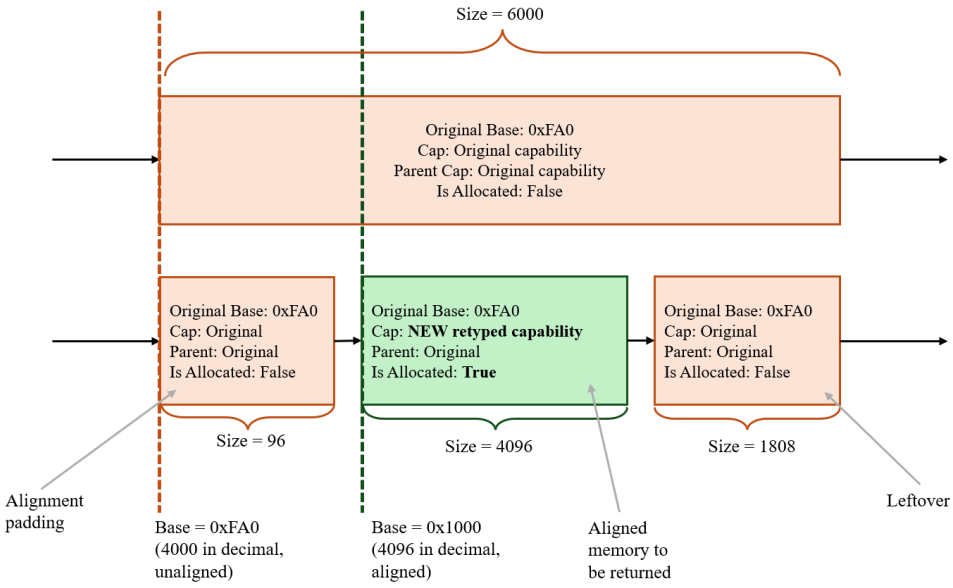

Figure 2: Example of Aligned Allocation

long term by distributing allocations more evenly across the address space.

3.2.3 Free

When freeing memory, the following checks are made:

1. Check if the capability is valid
2. Traverse the linked list to locate the corresponding memory node, verifying it by confirming the base address and size
3. If node is found, attempt to free it

When the node is found, it additionally checks that the node is marked as allocated, if it is, we begin the free. Once a node is freed and it's meta-data is updated accordingly, we attempt to merge the node with it's neighbours (previous and next nodes), if they are also free.

The merging operation, however, is restricted to nodes that share the same original base, meaning they originate from the same initial capability given during `mm_add`. This constraint exists because each retype operation requires a unified source capability, so merging across boundaries of different original capabilities is disallowed. By limiting merges in this way, we ensure compatibility with the retyping constraints.

We believe this merging strategy effectively reduces fragmentation in freed physical memory regions by combining adjacent free nodes into larger contiguous blocks. Without merging, each freed node would remain as a separate, smaller chunk, making it difficult to allocate larger contiguous blocks when needed. By merging adjacent free nodes that share the same original capability, we consolidate free memory into larger segments, and those segments can extended to ever larger if their neighbours are freed later, thereby increasing the likelihood of finding sizeable continuous memory regions in future allocations.

3.2.4 Partial Frees

When handling a partial free, we first check if the memory that is to be freed only partially matches the node. If yes, the node is split into up to 3 parts along where the memory matches the region that is being partially freed. Of these splits, up to 2 segments are still allocated and unreturned, and a single segment of the memory region is freed. If the memory region that is freed has any adjacent free neighbours, we attempt to merge them.

However, we acknowledge a drawback in this design: newly retyped capabilities (as shown in Figure 3) cannot be returned to the owner of the original retyped capability. After consideration, we believe a better policy might be to mark partially freed regions as "Partially Freed" and defer the destruction of the capability until the entire region is freed. This way, only when the full region becomes free is the capability destroyed, and the entire block is marked as available.

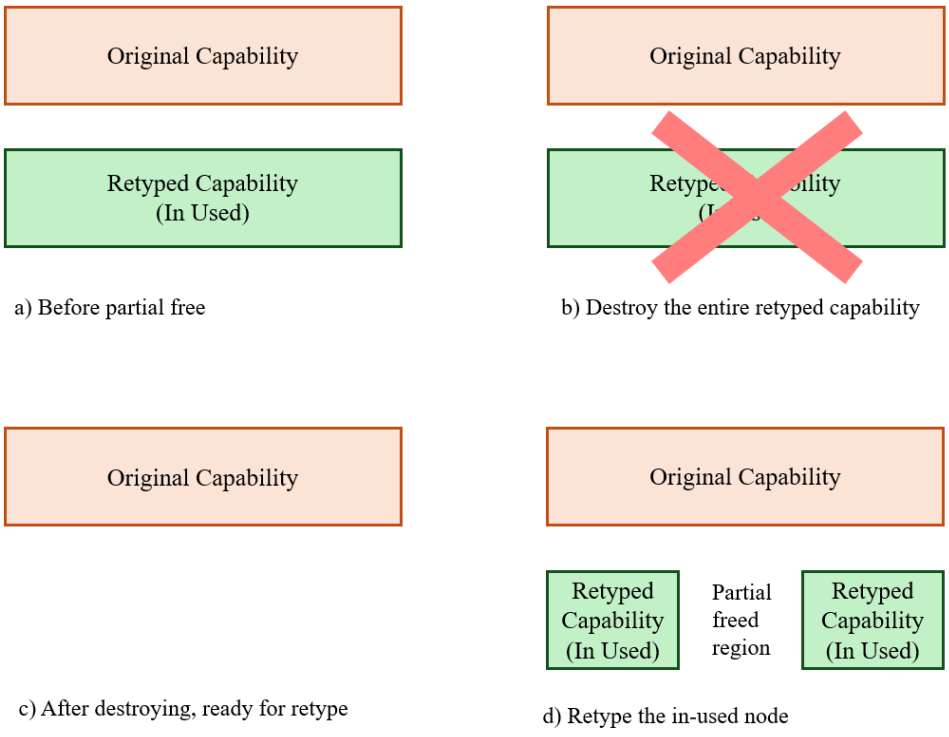

Figure 3: High-level Idea of Partial Free

3.2.5 Slab Refill

The data structures for the memory manager is stored in memory from a slab allocator, which relies on the memory manager in order to refill itself. In order to prevent a refill loop, first, refills are triggered early, leaving enough room in the slab to allow the creation of the meta-data to perform the refill. Second, a bit is set in the memory manager state to indicate that a refill is in progress, and that any allocations to the memory manager that happen while that bit is set are to refill the slab allocator, and should not re-trigger the refill process.

For this milestone, we set the slab refill threshold to four. This threshold is based on the typical case where a single call to `mm_alloc_aligned` may require up to two `mmnodes`: one for alignment padding and another for any leftover memory. Therefore, having at least two slabs available is generally sufficient. However, in rare cases, a `slab_refill` could trigger a `slot_refill` (due to additional CNode requirement), and `slot_refill` may call `mm_alloc`, which would need an additional two `mmnodes`. To account for this, we ensure that the slab count does not drop below four. This threshold is increased in later milestones.

3.2.6 Metadata

The meta-data for memory manager is stored in the memory manager instance. This includes:

1. A bit to indicate whether a refill is currently happening
2. The total amount of free memory remaining
3. The total amount of memory that has been allocated

3.3 Interaction With Other Components

The memory manager has the following interactions with these systems

1. Slab allocator
   Provides memory for metadata storage.

3.4 Limitations

Our straightforward design allows for efficient memory management that's easy to understand and maintain, which streamlines both implementation and debugging. While this simplicity offers clear benefits, there are some trade-offs in terms of performance.

Firstly, since it is a singly linked list, when finding a particular memory region corresponding to a node, we may have to traverse the entirety of the linked list in order to find the corresponding node. If there were n allocations, we would need up to $O(n)$ memory accesses in order to find the spot.

Secondly, since it is a singly linked list, once the node is found, additional traversals may be necessary in order to ensure that the newly created nodes maintain the linked-list invariant that nodes are linked in ascending order of base address, in addition to other checks such as overlapping memory regions.

Finally, the policy of our system is currently based on the assumption that only a single process is using the memory regions. When a memory region is freed, the policy is such that the region is freed for all processes that may have access to it.

4 Milestone 2

In the early stages of developing our virtual memory manager, we started with a very complex design that relied on a single, intricate data structure stored in our paging state. This design initially seemed appealing because, in theory, it could allow for great average-time performance. However, challenges arose, and we later pivoted to a simpler, easier-to-understand design. Although the final design may be less scalable in theory, it accomplishes what we intend.

4.1 Initial Approach

After milestone 1, we settled on a single **n-ary tree** data structure. This structure was meant to map allocated, unallocated, and lazy-allocated memory, capabilities, virtual address ranges, and page table slots. The approach attempted to mimic the structure of the physical page tables stored in memory, while minimizing the meta-data/data-structures needed to represent unallocated pages.

4.1.1 Data Structure

Each node had the following information:

1. The starting index of this page in the parent's page table (it can span multiple slots)
2. Capability for the slot it is in, in the page table
3. Mapping capability for this slot
4. Pointer to the next available node
5. Pointer to the previous available node
6. Children page tables (ex. For the L0, it this will be the L1 page tables)
7. The total number of pages available within this page's subtree
8. The number of slots the node represents within a page table
9. Flag for whether or not this node is lazy allocated

With this approach, we used the same data structure to track both the page table meta-data and the virtual memory space. The meta-data would be represented by nodes at layers 0-3, while the virtual memory would be represented as the children of the L3 nodes.

This approach had several benefits. For one, most lookups required at most 4 memory accesses to find the physical page. This was particularly useful for memory accesses, looking up physical pages, and freeing nodes. When working with memory allocations that fit within a single L3 page, this approach worked well because it generally also required at most 4 memory accesses. This was possible due to the meta-data that tracked how much space was available within a subtree—if there wasn't enough memory in a subtree, we wouldn't look at it.

4.1.2 Problems

There were several problems with this data structure that we had difficulties reconciling that ultimately lead us to decide to move onto a different data structure. Namely:

1. Virtual memory is continuous
2. Too complex to keep track of meta-data
3. Many edge cases

The most difficult issue to address was that virtual memory is continuous and can span multiple pages, such as L1, L2, L3, etc. This data structure made it trivial to allocate n pages within an L3 table, but spanning multiple L3/L2/L1 pages was very difficult, since by how the tree is structured, memory would naturally be split rather than be thought of as a "whole".

The complexity arose from the need to carefully track and read the meta-data in order to handle spanning multiple pages. When an allocation spanned multiple pages, the program would need to examine the individual node as well as its ancestors and other nodes at its level to determine if there was enough room spanning multiple pages. Once this was done, the node would need to be connected to all its ancestors, and when freed, the meta-data for all ancestors would need to be updated.

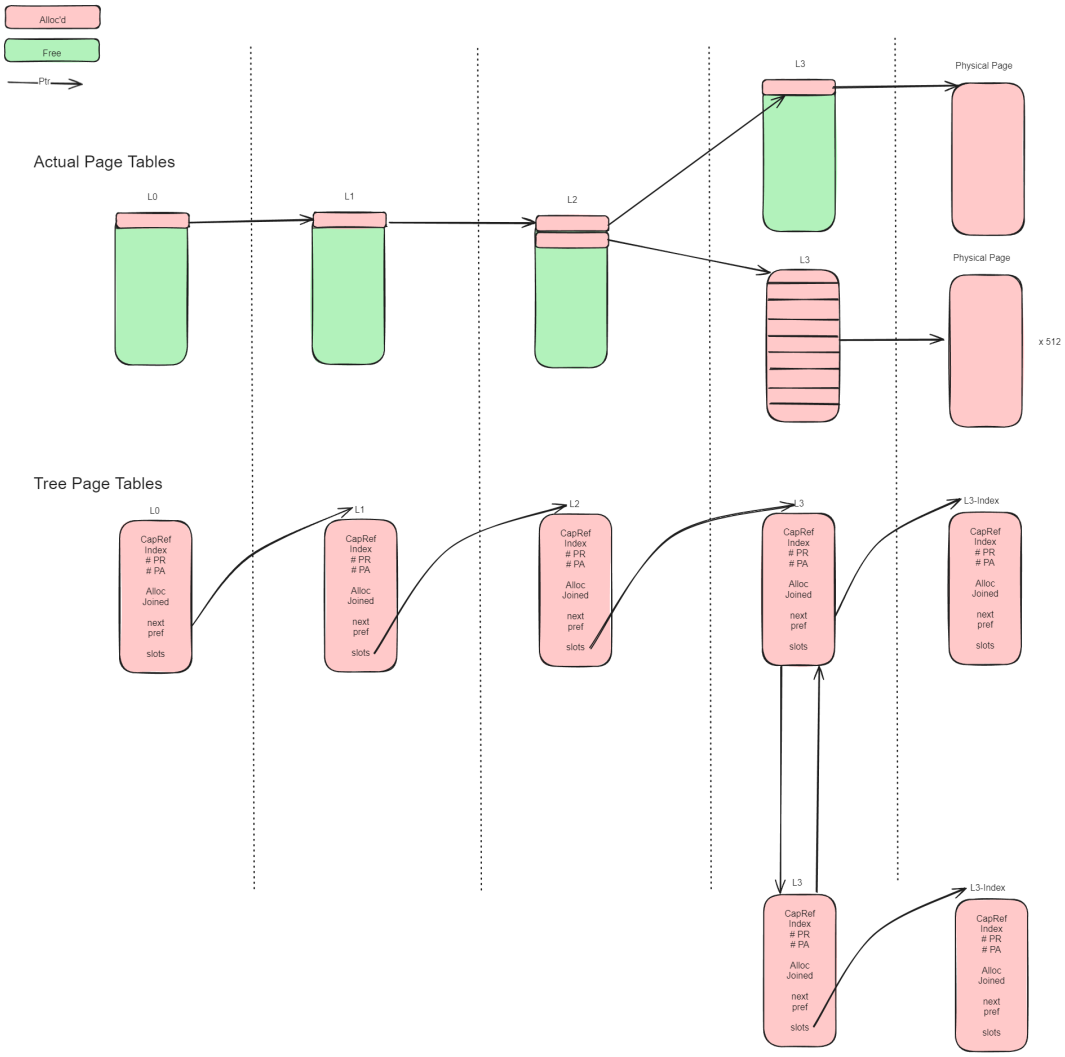

Figure 4: Initial Data Structure Tree

The updates to the meta-data were overly complex, and it was very easy to introduce bugs. As a result, we decided to abandon this data structure in favour of a simpler one.

4.2 Final Approach

The approach that we decided on used 3 data structures: one to track the virtual memory space, one to track the page table capabilities, and another to track the mapping capabilities. While on paper the performance of this method should be worse, it makes up for it in being easy to understand and easier to bug fix.

4.2.1 Data Structure

Our virtual address data structure was a single doubly-linked list that represented the entirety of our allocated/lazy-allocated virtual memory space. In this data structure, a node of the linked list keeps track of:

1. The starting virtual address that this node represents
2. The initial virtual address (for a group of nodes)
3. The size of the frame of allocated memory that the node represents
4. A flag for whether or not the memory has been allocated, or just lazy allocated
5. A pointer to the data structure that stores the mapping capabilities for the frame, into their respective L3 tables

Compared to the previous data structure, this made it much easier to find areas of memory that was available to be allocated, as everything is continuous. It was also easier to do

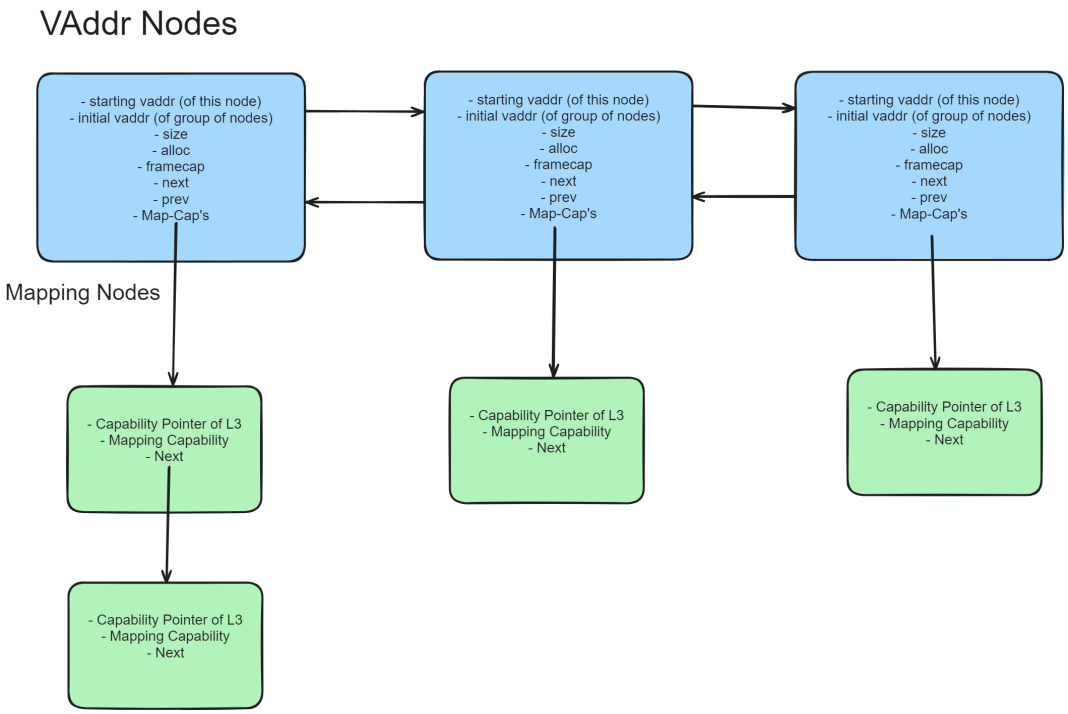

Figure 5: Data structure to track virtual address space, and mapping capabilities

lazy-allocation with this data structure - we only allocate the page that is currently being accessed, instead of the entirety of the allocated memory. Additionally, any gaps between virtual addresses between two nodes of the linked list are considered to be free - not allocated and not lazy allocated. The initial virtual address that is stored within each node allows us to tell which is and isn't part of "the same area of allocated memory".

Within these nodes, we also keep a reference to our second data structure, which is a singly linked list that holds a reference to the L3 table that the frame is mapped in, as well as the mapping capability within that table. Using this, we can quickly de-allocate a node from both the virtual address linked list and the actual page tables.

Our page table related capabilities were stored in a separate data structure, which was once again an n-ary tree. The data structure had:

1. The capability of this page table
2. The mapping capability of this page table in it's parent's slot
3. The index that this page table is in, in it's parent
4. The size (currently unused, for super pages)
5. An array of pointers to all 512 of it's children

This data structure made it trivial to look up the capabilies required to allocate a frame - just decompose the virtual address into page table indices, and traverse the tree. Since every node has an array of pointers to it's children, it is also trivial to allocate new page tables, we simply check if a given slot is null, if yes, create a new page table.

While this data structure does accomplish our goals, it is evidently limited by the doubly linked list data structure for finding available virtual addresses. When allocating or freeing, there is a $O(n)$ cost of finding the node, where n is the number of all things that have been allocated to virtual memory.

4.3 Algorithms

This is a high-level overview of the different algorithms that are used for paging.

4.3.1 Alloc

For both `paging_map_frame_attr_offset` and `paging_map_fixed_attr_offset`, an availabe virtual address must first be found, and then mapped.

Finding Available Addresses

Finding virtual addresses is done via first-fit. When finding an available address we traverse the vaddr linked list until we

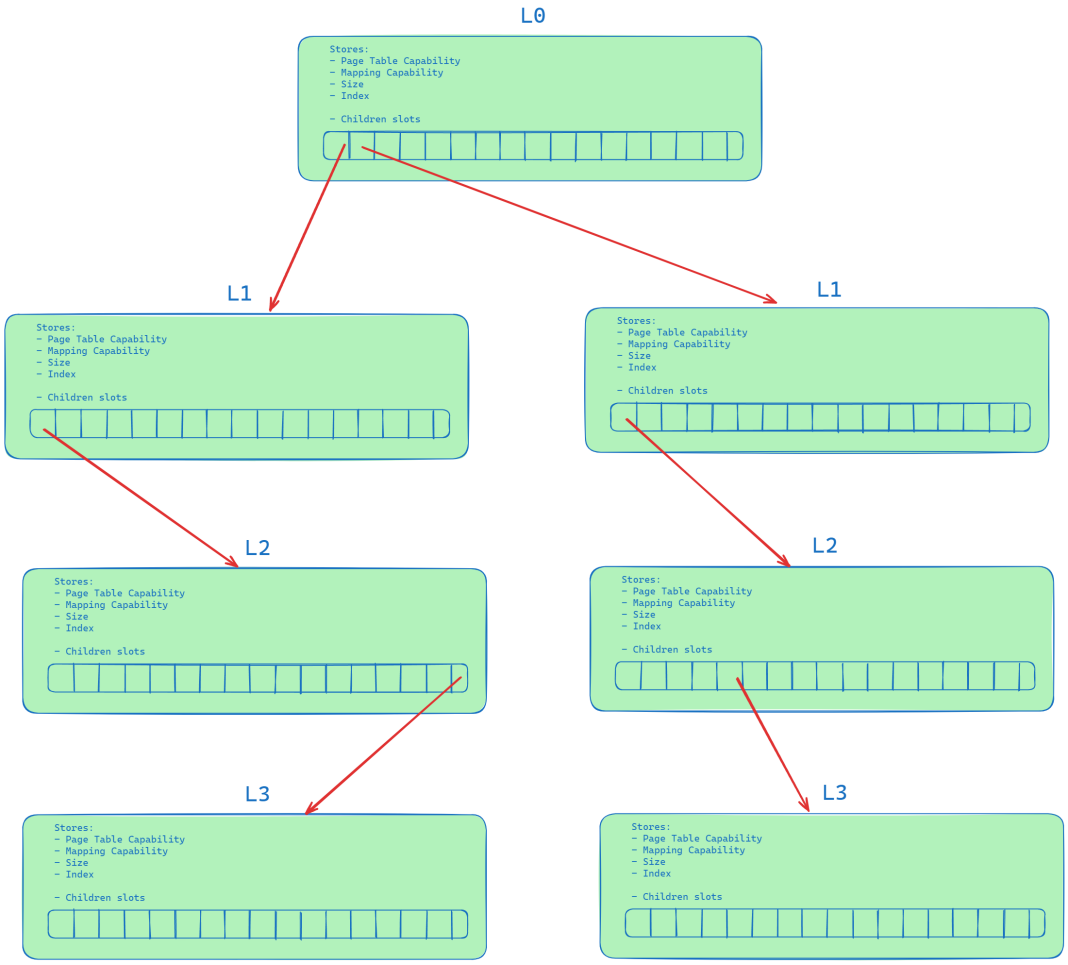

Figure 6: Data structure to track page table capabilities

find a gap. We then do the following checks:

1. Check if it is large enough to support the size of the frame that we are trying to allocate
2. If we are trying to allocate to a specific address, see if the address is within the gap
3. If there needs to be special alignment, see if the size of the gap is large enough to account for alignment padding

Mapping Addresses

When mapping an address to a frame, we do the following steps using the virtual address, and the size of the data to allocate (in pages):

1. Start at L0 in Tree data structure. Parse virtual address for L1 index.
2. If L1 hasn't been allocated, allocate it and link to L0
3. Lookup L1 slot, parse virtual address for L2 index
4. If L2 hasn't been allocated, allocate it and link to L1
5. Lookup L2 slot, parse virtual address for L3 index
6. If L3 hasn't been allocated, allocate it and link to L2
7. Map frame to L3. We map either the minimum of the size of the frame in pages, or the number of pages remaining in the L3
8. If the frame still hasn't been fully allocated, update virtual address + offset based on the number of pages allocated, and repeat from step 1.
   Ex. We were allocating a frame of size 10 pages at address 0x1, and we were only able to allocate 1 page. We would then recurse, attempting to map 9 pages at virtual address 0x2

Note: As this is done internally by a helper function, it is only called after an virtual addrss is found, and thus we

assume that the virtual address provided to be mapped is free.

Lazy Allocation

`malloc` and `paging alloc` are examples of where we do lazy allocation. When doing lazy allocation, when requesting a virtual address of size *n*, we will will follow the same steps as with finding an available virtual address, but instead of immediately mapping a frame to back the allocation, we set the lazy allocated flag to true.

When a page fault occurs or someone tries to map to an address (using `paging_map_fixed_attr_offset`) that is within a lazy allocated region, we split up the node, and only allocate the page that is being accessed. The new nodes that are created from the split will all share the same initial virtual address, so when it comes time to free or join nodes we can tell what is grouped.

`malloc`

`malloc`, and morecore work with the following. Upon initialization the minimum alignment is saved in the current morecore state.

In our system, `malloc` lazy-allocates an arbitrary memory region, rounded up and aligned to the minimum alignment. This is done with the `paging_alloc()` function. `malloc` returns memory regions arbitrarily from `paging alloc`, rather than from a memory pool like the `slab` allocator. This choice was simply made out of simplicity.

Page Faults

When a page fault occurs, the following happens:

1. Check if the fault is a page fault
2. Check if the virtual address is valid. This means that it is within the user region of memory, and not a null pointer (Virtual address 0x0)
3. If that is the case, we find the virtual address node. If the node hasn't been lazy allocated, this is also an illegal access and we error
4. Once the node has been found, we check if it has already been allocated. If it has, this is also illegal, and we error
5. Finally, once those checks pass, we allocate the page that has been accessed. If the node representing the memory region is larger, it is split

4.3.2 Free

When freeing a node, we first find it within the virtual address linked list. Once the node has been found, we deallocate the frame using the mapping capabilities and the page table reference stored within the mapping node. We then clean-up and free the mapping node linked list, and then we clean-up and free the virtual address node.

If a virtual address node has been split due to lazy allocation, we will free those as well.

4.3.3 `slab` Refill

Since `slab` refilling ultimately relies on the virtual memory manager, special care is taken to avoid an infinite refill loop, where an allocation triggers a `slab` refill, which then triggers another allocation, and so on.

To avoid this, we introduce a threshold to ensure enough space is available for all necessary allocations in the virtual memory manager when refilling the `slab`. Additionally, we set a flag to indicate that a refill is currently in progress, preventing the allocations within a refill from retriggering the refill process.

Multiple data structures may trigger a refill at any time, including the one in the memory manager (MM). A special edge case occurs when creating metadata to track the mapping of a new page table. In this case, the refill can interrupt the current allocation process, and while refilling, may map the new page table before the original program does. Therefore, before any page table allocations, we need to check if the tables have already been allocated, as they might be allocated during our attempt, ensuring synchronization.

4.4 Interaction With Other Components

The paging and virtual address systems interacts with many key components across the project, relying on several dependencies to support memory management and address allocation:

1. MM (Memory Manager)
   For the management of the ram capabilities used for tracking meta-data, and creating frames
2. Morecore
   For integration with `malloc`/`free`, which use the paging functions to lazy-allocate and free memory
3. `slab` allocator
   Provides memory for metadata storage, and paging systems are used to refill the `slab` when it fills up, without causing a page fault.

4.5 Limitations

There are several limitations within the current system.

First, as previously mentioned, traversing the linked list is costly. For allocation or deallocation, there is an $O(n)$ cost to locate the node, where n is the number of items allocated to virtual memory. This implies that over the system's lifetime, as memory allocations increase, the cost of new allocations and accesses will rise accordingly.

Second, individually allocated memory regions can become fragmented if they were allocated lazily. Currently, memory is accessed one page at a time. For a memory region of size n pages, if every page is accessed, the region will be split into n nodes, further increasing the linked list size and degrading performance. Performance could be improved in the future by merging nodes within a single allocated region.

Third, page tables, once created, are not destroyed. This policy avoids the overhead of frequent table creation and destruction, as well as potential synchronization issues. However, it also means the system may "leak memory," or maintain a larger runtime footprint than necessary. Future improvements could include periodic garbage collection to clean up unused tables.

Finally, the data structures for managing the virtual memory space are relatively large. Specifically, the page table capability tree has considerable overhead, as it includes pointers to all 512 of its children. This is particularly impactful for the L3 page tables, where many pointers remain unused. Over the system's lifetime, this could lead to significant memory consumption.

4.6 Retrospective and Improvements

During this milestone, our team gained significant insights into software design and decision-making. The tree data structure, initially implemented in Milestone 1, was a component we were invested in due to its initial success and alignment with our project goals. While it performed well and passed many tests, the emergence of increasingly complex edge cases exposed its limitations. Over time, the implementation became overly intricate, bloated, and challenging to maintain or reason about effectively.

Ultimately, we made the decision to pivot and abandon the existing design. This experience underscored an important lesson about the sunk cost fallacy: in engineering and design, it is sometimes more effective to pause, critically evaluate the current approach, and decide whether to iterate or start anew. This milestone highlighted the value of adaptability and the importance of prioritizing simplicity and maintainability in complex systems.

A tree-based structure, or a similar hierarchical data structure, presents the most promising approach for overcoming the current limitations in our design, particularly in efficiently managing address availability and allocation. By implementing a tree structure, we can achieve faster lookups, insertions, and deletions for memory addresses, as it naturally supports hierarchical relationships and enables efficient segmentation of the virtual memory space.

5 Milestone 3

Compared to the memory and virtual memory management systems described in previous milestones, process spawning offers less flexibility in design choices but demands precise and correct implementation. The process-spawning function must be accurate to ensure that each child process is equipped with all necessary resources to initiate successfully.

5.1 Spawning

5.1.1 Data Structure

We begin by describing our `spawninfo` structure, which serves as a management and tracking tool for the state and resources involved in the spawning process. Beyond basic metadata such as the binary name and process state, `spawninfo` includes fields dedicated to managing the child process's capabilities and virtual memory. These fields provide references to critical components like the child's root CNode, task CNode, page CNode, dispatcher handle, entry point address, memory manager, paging state, etc., along with a slot allocator for efficient capability allocation within the child's CSpace. The information encapsulated in `spawninfo` enables us to modularize the spawning steps into distinct functions while ensuring access to essential resources at different stages of the spawning process.

5.1.2 Elf Parsing

Initializing the child process begins with parsing the elf image. This is done by initially finding the required module from multiboot image, mapping the elf image into the parent's address space, and parsing to verify that the magic bytes are present. If the elf image is valid, we continue to attempt to create the child.

5.1.3 Set-up CSpace

In this section, we introduce the setup of the Capability Space (CSpace), a critical component for managing access to resources in the Barrelfish operating system. The primary objective of setting up CSpace is to ensure that the child process has organized access to the necessary resources,

enabling it to spawn and execute successfully.

Creating Well-known CNodes We first begin by creating `L2CNode`s in defined, well-known locations in the `L1CNode`. This includes:

1. Top level page table
2. `Root Taskcn`
3. `Root Alloc_0`
4. `Root Alloc_1`
5. `Root Alloc_2`
6. `Root Pagecn`

After, we create the following capabilities in the `Taskcn`:

1. Selfep
   (a) This is created by retyping the dispatcher capability to `ObjType_EndPointLMP`
2. Rootcn
3. Dispatcher
   (a) Created with a frame of size `DISPFRAME_SIZE`
4. Dispframe
   (a) Created with a frame of size `DISPFRAME_SIZE`
5. Argspage
   (a) Created with a frame of size `ARG_SIZE`
6. Earlymem
   (a) Created with a ram capability large for bootstrapping memory allocations (currently defined as 512 pages)

5.1.4 Set-up VSpace

In this section, we discuss the creation of the virtual address space for a process in our operating system. The main goal of setting up the virtual address space is to ensure that each process has its own isolated memory region, by setting up an appropriate child paging state

The virtual address space is initialized in the following steps:

1. The child's L0 page table capability is created, and mapped to both the child CSpace, as well as the parent. The parent requires the child's L0 page table to write into the child's virtual address space.
2. The child's paging state is initialized, and flagged as a foreign paging state

5.1.5 Elf Loading

Once the virtual address space is established, the ELF is loaded using `elf_load` with an allocated ELF allocator. The ELF is allocated to a fixed address in the child process and to an arbitrary location in the parent.

We begin by determining the size required to map the ELF image, taking into account any additional space needed to accommodate the fixed address in the child process, as the address may not be page-aligned. Once the required size and base address are determined, the ELF is mapped to the child process's virtual address space at the fixed location. Subsequently, it is mapped to the parent's virtual address space, with the return address adjusted to account for any alignment updates that were necessary.

5.1.6 Set-up Dispatcher

Next, the dispatcher is created. To create the dispatcher, a frame is first allocated and mapped into the parent's virtual address space to store the child's dispatch frame. The child dispatcher is then created and mapped into the child's virtual address space.

Once this is complete, the dispatcher's enabled and disabled register save areas are updated with initial information to prevent the process from immediately crashing. This information includes:

1. Core id
2. Domain id
3. `Udisp`
4. Disabled Flag
5. Name (for debugging)
6. Starting Address for the program counter in the disabled area
7. Initialize offset registers

5.1.7 Set-up Environment

Finally, this section details how the environment is set up.

Set-up Arguments

Arguments for the process are parsed and processed upstream from this point, either passed via `spawn_load_with_caps` or parsed from the multiboot command line. These arguments are stored in the `spawn_info` structure and are retained until the completion of prior setup steps, including configuring

capabilities, establishing the virtual address space, loading the ELF, and setting up the dispatcher.

To pass the arguments from the parent to the child, the parent must write the arguments into the child's address space. This is accomplished by mapping the child's Argument Space frame into both the parent's and the child's virtual address spaces, and then modifying it from the parent's space. This allows direct modification of the frame, enabling the transfer of arguments to the child.

Passing Capabilities

To allow the parent process to pass additional capabilities to the child process at start-up, we faced the challenge of ensuring the child could locate and use these capabilities after spawning.

Initially, we attempted a simpler solution by using the child's slot allocator to assign slots and then copying the capabilities directly into these slots. However, although the capabilities were stored in the child's CSpace, the child process had no knowledge of their locations or quantity, rendering them inaccessible.

Our final approach introduced a metadata frame in slot 1 of the page CNode. This frame was created immediately after setting up the child's VSpace to ensure that slot 1 would always be available and consistent. We treated this frame as an integer array: the first entry stored the slot of the first capability (in the page CNode), and the second entry recorded the total number of passed capabilities. The child process could then map the frame from slot 1 in the page CNode, read the metadata, and access the capabilities as needed. We verified this approach by having the parent pass RAM capabilities to the child, which the child could retype to frames, map, and access successfully.

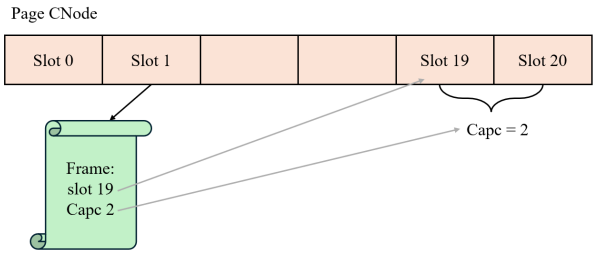

Figure 7: Passing Capabilities During Spawning

5.1.8 Start, Kill, Resume, and Suspend

For the following process functions, we simply check if the process is in the correct state, and if it is, we call the corresponding `invoke_dispatcher_*` function, and update the spawn info struct with the new state if successful.

Note that the handlers for the `invoke_dispatcher_*` functions for Kill, Resume, and Suspend operations are also implemented by us. The Kill and Suspend actions are accomplished by removing the dispatcher from the run queue, which is organized as a linked list under the hood. In contrast, the Resume action works by adding the dispatcher back to the queue. Notably, Kill and Suspend involve similar operations, but they are differentiated by setting the `spawn_info` structure to different states. Only a `spawn_info` in the "Suspend" state is permitted to be resumed in the future.

5.2 Process Management

5.2.1 Data Structure

The list of current processes are stored in a doubly linked list. In the doubly linked list, the following meta-data is tracked:

1. Pointer to spawn info struct for the process
2. The name of the process (up to 128 characters)
3. The length of the name of the process
4. The core id that the process is running on

The choice of a doubly linked list was made due to its simplicity and efficient node removals, which may occur frequently. Including the length of the name, in addition to the name itself, provides a more efficient initial check for name equality before performing the final comparison.

Meta-data related to the processes themselves is not stored in the nodes, as it can easily be obtained from the `spawn_info` structure. The `spawn_info` structure is stored in the node rather than a process status node because it is straightforward to convert a `spawn_info` structure to a process status node, and the `spawn_info` is already required for auxiliary functions like suspending a process.

When creating a new process is created, it has it's node created with the relevant meta-data, and then it is appended at the front of the list.

PIDs

During this milestone, PIDs are assigned using an incrementing counter stored in a global variable. Each time a new process is spawned, it uses the current value of this counter and increments it. Later on in milestone 6, we would change this to have the most significant bit be used to indicate which core the process is running on.

Roads not taken

The choice to use a linked list was purely due to it's simplicity, and similarities with other parts of the system, but comes at the cost of performance. There were some other roads that were considered, but not followed through on:

1. **Red-Black / AVL Tree**

   Using an AVL tree is an appealing choice because they are also relatively simple data structures. With an AVL tree, the value/weight of a node would simply be the PID, and given the characteristics of a AVL tree, would allow it to have an average insertion, removal, and retrieval cost of $O(\log n)$. However, given how newly added pid's will only ever increase, this data structure will frequently have to re-balance itself, which may degrade the potential performance gains.

   Thus, given that with how our PID's are allocated, and the added complexity of implementing an AVL tree, we decided not to pursue this option.

2. **Hash Table / Hash Map**

   Hash tables were also an appealing choice. A potential design could have used the PID and the meta-data struct as the key and value respectively in a key value pair. Depending on library implementation, this could give an average constant time read, write, removal ($O(1)$). Unlike trees, most library implementations wouldn't have their performance degraded by our PID allocation policy as well.

   The difficulty for hash tables, and why they were not pursued, was because of the difficulty to iterate through all keys/values within the data structure. Most of the libraries that we explored, and the one provided in the Barrelfish repository, make this not as trivial as in other higher-level languages. Thus, if we were to use a hashmap, the operations to find all running processes, or processes with a specific name become much harder.

5.2.2 Algorithms

The algorithms used for process management are relatively simple compared to other components of the operating system.

When obtaining a process by PID or name, the linked list is traversed until the process with the matching meta-data is found, at which point it is returned. Similarly, for commanding processes, the list is traversed until the process with the matching PID is found, and the corresponding command is issued. If the process is killed and the operation is successful, it is removed from the list.

When aggregating running processes, the linked list is first traversed to count the number of running processes, in order to get the number of running processes, and allocate enough memory to store the list of processes. The list is then traversed a second time to add the running processes to the list.

5.3 Interaction With Other Components

The process spawning interacts and uses many different parts of the system. In particular, it interacts with:

1. MM (Memory Manager)
   The memory manager is used to create the new memory manager for the child process. This is also used under the hood for the management of the many frames that are created to be passed to the child

2. Paging Systems
   The paging system is used, in conjunction with higher-level abstractions like malloc to allocate memory for meta-data. In addition, paging functions are used directly to be able to map elf's to specific addresses, or to allocate and map frames directly without having to pagefault.

5.4 Limitations

Since there were fewer design decisions made for process spawning, the limitations focus primarily on process management.

First, PIDs cannot be reused sustainably. Once a PID is allocated, it cannot be reused by another process until the system is restarted, as it is a global, incrementing variable. While the system is unlikely to run out of PIDs, given that it would require spawning $2^{64} - 1$ processes, this design also prevents spawning a process with a predetermined PID.

Second, the linked list data structure requires multiple memory accesses to look up processes, which becomes a concern as more processes are spawned. Additionally, since earlier processes typically finish first, and new processes are appended to the front of the list, this could lead to an expensive remove operation.

Third, the choice of 128 character max process name was arbitrary, it may be beneficial in the future to instead allow for arbitrarily long names, stored on the heap

Finally, aggregating the list of currently running processes requires two traversals of the list: one to count the processes and another to allocate and map them. An alternative approach could involve tracking the total number of processes managed,

regardless of their state, and allocating excess memory to eliminate the need for a second traversal. This would provide more than enough space, but at the cost of precision.

5.5 Retrospective and Improvements

During this milestone, our team developed valuable insights into the complexities of software design and debugging. The process of spawning a new process proved to be intricate, with numerous small bugs emerging that hindered successful spawning. This experience highlighted the importance of debugging within complex systems, particularly when dealing with components that operate as "black boxes". It also underscored the criticality of writing code that is both maintainable and easy to debug.

Moreover, this milestone provided a practical lesson in task division. Despite the numerous dependencies between different system components, once the design was solidified, we were able to effectively partition tasks for independent functions, including parts of the initialization process and the process management wrapper. This approach not only facilitated smoother implementation but also enhanced our ability to isolate and resolve issues efficiently.

As for future improvements, the change that would have the most impact would be to migrate from the linked list approach to storing running process metadata to one of the alternative data structures suggested in our report. Unlike with memory management and virtual address space management, the sequential nature of a linked list does not have any intrinsic benefit design wise, other than being simple. If in the future performance becomes an issue, this would be the area that would be explored.

For future improvements, the most impactful change would be transitioning from the current linked list-based approach for storing running process metadata to one of the alternative data structures discussed in this report. Unlike memory management and virtual address space management, where the sequential nature of a linked list offers certain design advantages, the linked list does not provide any intrinsic benefits in the context of process management, other than its simplicity. Should performance become a concern as the system scales, this area would be a primary focus for optimization. By adopting a more efficient data structure, such as a hash table or balanced tree, significant improvements in process lookup, insertion, and deletion times could be realized, thus enhancing overall system performance.

6 Milestone 4

In this section, we detail the implementation of inter-process communication (IPC) within a single core using Remote Procedure Call (RPC) built on top of Lightweight Message Passing (LMP) channel. Developing a fully functional LMP-based RPC system is a complex process, requiring careful attention to nuances such as multithreading safety. To manage this complexity, we began with a simple and minimal design, incrementally refining and expanding it to achieve a robust and efficient solution.

6.1 Data Structure

The `aos_rpc` structure serves as the core data structure for this milestone, encapsulating the necessary information for a child process to communicate with the `init` process. Designed for simplicity and clarity, it includes the following fields:

1. **LMP Channel**: The communication channel established for the child process to interact with the `init` process.
2. **Waitset**: The default waitset associated with the channel, used for managing asynchronous communication.
3. **Thread Condition Variable**: Ensures multithreaded safety for RPC operations; further details are provided in Section 6.15.
4. **RPC Initialization Boolean**: A flag indicating whether the handshake with the `init` process has been completed (i.e., the handshake has been sent and the acknowledgement received). This is discussed further in Section 6.4.

6.2 AOS RPC Set-up

Before sending messages between processes, we must first set up the LMP channel for child processes. After the spawning process completes—but before the dispatcher is invoked to make the child process runnable—the `init` process creates an LMP channel specifically for **THIS** child process. During this step, a new `init` local endpoint is created using `cap_selfep` as a template, with a dedicated buffer minted for the endpoint. The `init` local endpoint capability is then copied into the child's CSpace at a well-known slot, enabling the child process to establish its LMP channel.

Once the child process begins execution, it creates its own LMP channel, mints buffer to its local endpoint, and uses the `init`'s local endpoint (stored in a well-known slot in its CSpace) as the channel's remote endpoint. A handshake is required to make the channel bidirectional, allowing `init` to obtain the child process's local endpoint (detailed in Section 6.4). This setup ensures that each child process has a dedicated bidirectional LMP channel with the `init` process, with endpoint capabilities exclusively bound to their respective channels, thereby avoiding communication interference between `init` and multiple child processes.

Road Not Taken

At the beginning, we planned to establish only one LMP channel from `init` to **ALL** child processes. This would be achieved by using a global variable to track whether the LMP channel had already been set up for `init`. Once the channel was created after spawning the first child, no additional LMP channels would be created for any subsequent child processes (meaning that only one `init` local endpoint will be created).

In this design, one of the critical points was ensuring that `init` could get the correct child process' endpoint it needs to talk to. To accomplish this, we intended to extend the process management data structure from the previous milestone to include each child's local endpoint. During the handshake, a child process would send its PID and local endpoint to `init`, which would store the information in the process management node corresponding to that PID. For subsequent messages, the child would include its PID, allowing `init` to retrieve the correct node and bind the appropriate endpoint for replies.

This design was discarded for two main reasons:

1. It introduced an additional layer of traversal during message sending, unnecessarily increasing complexity and degrading performance.
2. Using a single `init` local endpoint for all child processes created a higher likelihood of buffer management issues. Since the buffer is tied to the endpoint, simultaneous message sending from multiple child processes could result in buffer conflicts.

6.3 Overall Design

This milestone implements several RPC functions, most of which follow a similar pattern. In this section, we provide a brief overview of the general flow of the RPC implementation from the perspectives of both the `init` process (acting as the server) and the child processes (acting as clients). Detailed descriptions of each RPC function are provided in the subsequent sections.

6.3.1 Channel

We opted to use a single channel between `init` and each child process for all types of RPC requests, meaning that we do NOT distinguish between separate channels for `init`, memory, process, and serial communications.

We believe that this approach is simpler, more manageable, and reduces the likelihood of multithreading issues arising from managing multiple channels.

6.3.2 Message Structure

We begin by describing the data structure used for RPC messages. Sending a message through an LMP channel requires one of nine functions, `lmp_chan_sendX`, ranging from `lmp_chan_send0` to `lmp_chan_send8`, each capable of sending X words of payload, where each word is 64 bits. To accommodate this, we carefully select the appropriate number of words (X) and treat the payload of each message as a structured data format comprising the following fields:

1. **Type of the RPC message:** The first word of the LMP payload specifies the type of the RPC message. This includes function identifiers and acknowledgements (e.g., `AOS_RPC_HANDSHAKE`, `AOS_RPC_HANDSHAKE_ACK`, etc.), which will be introduced in the following sections.
2. **RPC message-specific payload:** Depending on the RPC type, this payload occupies 0 to 7 words and contains data relevant to the function (e.g., a number for `aos_rpc_send_number`).

6.3.3 Child Processes as Clients

Child processes act as clients, requesting services from the `init` process via RPC function calls. The complete workflow of an RPC function call consists of two main components: `aos_rpc_*` and `aos_rpc_*_recv_handler`. In our design, each `aos_rpc_*` function has its own dedicated `aos_rpc_*_recv_handler`, waiting for the acknowledgement specific to that function.

aos_rpc_*

The `aos_rpc_*` functions serve as both the **messaging layer** and the **transport layer**. These functions generally consist of the following key steps:

1. Pack the message for transmission.
2. Allocate receive slot if necessary.
3. Register the corresponding receive handler as a closure. This closure includes the receive handler and an argument, which must contain at least the `aos_rpc` structure. If the associated `aos_rpc_*` call requires a return value to be written to a pointer, the pointer is also encapsulated within the argument.
4. Send the message through the LMP channel.
5. Wait on the default waitset for the channel.

aos_rpc_*_recv_handler

Once a message arrives at the LMP channel, an event is raised in response to the activity, triggering the receive handler

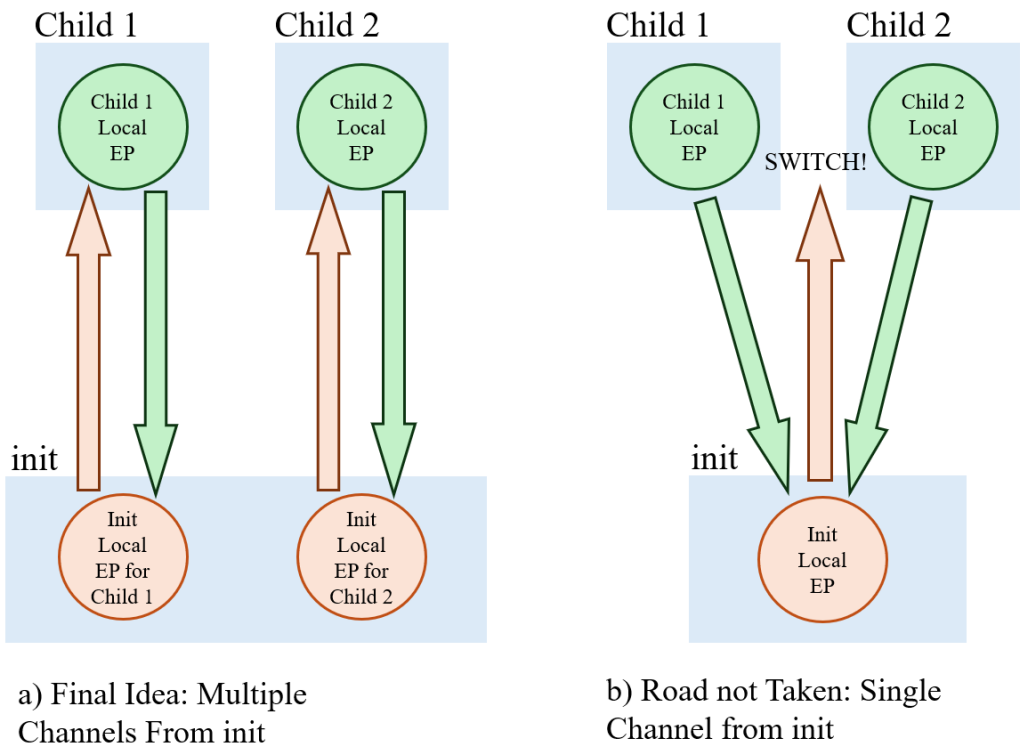

Figure 8: LMP Channel Initialization Comparison

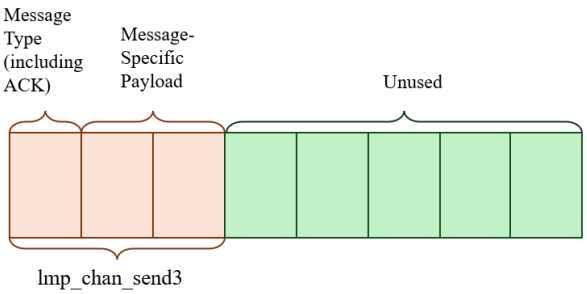

Figure 9: Message Structure Example

registered in the previous step. On the client side, the receive handler typically includes the following key steps:

1. Unpack the argument passed to the receive handler and retrieve the `aos_rpc` structure, which contains the LMP channel.
2. Receive the message from the LMP channel.
3. Verify the acknowledgement type in the received message.
4. If the function requires a return value, unpack the return pointer from the argument, extract the return value from the received message, and write the value to the pointer.

Note that the receive handler is automatically de registered after it is triggered. On the client side, the receive handler is not re-registered until the next `aos_rpc_*` function call. This design ensures that the channel is associated with a dedicated receive handler for each specific use case, allowing messages to be processed as needed.

6.3.4 Init Process as Server

The `init` process acts as a server, continuously listening to channels and waiting for incoming messages. When a message arrives, a centralized server receive handler is triggered to process various types of requests sent by clients.

recv_handler

The `init` process's receive handler is implemented as a large switch statement that receives and unpacks the message, switching on the first word of the message payload (see Section 6.3.2). For each case in the switch statement, the receive handler performs the following steps:

1. Processes the incoming request based on its type.
2. Packs the acknowledgement message.

3. Sends the acknowledgement message back to the client.

Unlike the child process's receive handler, which is specific to each `aos_rpc_*` function and de registers after being triggered, the `init` process re-registers its receive handler each time it is invoked. This ensures that the `init` process is always ready to handle incoming messages and respond to any type of request. This distinction reflects the different roles: the child process (as a client) sends requests as needed, while the `init` process (as a server) remains continuously available to respond.

6.4 Handshake

The handshake is used to allow the child process to send its local endpoint to the `init` process. When `init` first sets up the LMP channel for the child, the child process has not yet started running. As a result, `init` does not have the child's local endpoint and must initially set the remote endpoint of the channel to `NULL_CAP`. Once the child process starts running, it performs the handshake by sending its local endpoint to `init`. This enables `init` to bind the child's local endpoint as the remote endpoint of its LMP channel, completing the bidirectional channel setup.

In our current design, the `aos_rpc` structure is initialized with the "RPC Initialization Boolean" set to `false` (as described in Section 6.1). During the first `aos_rpc_*` call, the child process performs a handshake, which also sets the boolean to `true`. This ensures that subsequent `aos_rpc_*` calls bypass the handshake step.

The client message consists of the following fields: a message type (`AOS_RPC_HANDSHAKE`) and a child's local endpoint capability.

The server message consists of the following field: an acknowledgement type (`AOS_RPC_HANDSHAKE_ACK`).

6.4.1 Road Not Taken: Where to put handshake?

Initially, we believed it made sense for a child process, upon being spawned and starting execution, to immediately send a handshake to its parent to establish the bidirectional LMP channel. As a result, we embedded the handshake process within the initialization of `aos_rpc` (mentioned in Section 6.4).

However, this approach caused issues during the M3 test script. In the spawning process milestone, child processes are designed to run without requiring inter-process communication. The test script exits the system before the `init` process begins listening to the waitset. This created a problem because the child process automatically initiated the handshake during initialization, even when no communication was necessary. Consequently, the acknowledgement could not be received, and no child process proceed.

6.5 Send Number

Sending a number is the simplest form of an RPC function call. The client sends a number to the server, the server echoes the received number, and then sends an acknowledgement back to the client.

The client message consists of the following fields: a message type (`AOS_RPC_SEND_NUMBER`) and a number.

The server message consists of the following field: an acknowledgement type (`AOS_RPC_SEND_NUMBER_ACK`).

6.6 Send String

Due to the limitation of `lmp_chan_sendX`, where at most eight 64-bit words can be sent at a time, longer strings must be broken into smaller pieces. In our current design, the string is divided into chunks that fit within the word limit. Each chunk is packed and sent in a loop until the end of the string is reached. On the server side, the chunks are unpacked, reconstructed, and the string is displayed.

The client message consists of the following fields: a message type (`AOS_RPC_SEND_STRING`), a chunk size indicating the length of the current chunk, and up to six words of payload.

The server message consists of the following field: an acknowledgement type (`AOS_RPC_SEND_STRING_ACK`).

6.6.1 Drawback

We acknowledge that this design performs well for small strings that can be delivered in a single message. However, it is less effective for large strings. The server (`init` process) treats each chunk as an independent message and displays them separately, rather than recognizing them as parts of a single string, which may not align with the intent of the client process.

6.6.2 Possible Fix: Sending Large Strings

Although sending large strings is not currently supported in our system, we collaboratively devised some feasible plans to address this limitation. Two potential approaches were identified as relatively straightforward to integrate into our existing system:

1. Create a string buffer for each thread on the `init` side.

This approach involves extending the process management linked list to include an additional linked list, where each node corresponds to a thread. When a thread within a child process sends a large string, it includes additional metadata alongside the current fields (type and chunk size): its PID, thread ID, and a flag indicating the end of the string.

Upon receiving a part of a large string, the `init` process traverses the process management list to locate the appropriate thread buffer. The payload is then accumulated into this buffer. Once the buffer receives the chunk marked with the end-of-string flag, the `init` process reconstructs the full string, displays it, and cleans up the buffer to prepare for future messages.

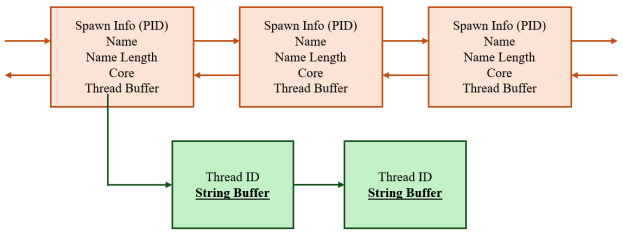

Figure 10: Large String with Thread Buffer

2. Send a frame capability.

In this approach, the child process allocates and maps a frame into its own address space, writes the string into the frame, and then sends the frame's capability to the `init` process. Upon receiving the capability, the `init` process maps the frame into its virtual address space, reads the string from the frame, displays it, and unmaps the frame when finished.

6.7 Serial RPC

The `serial_getchar` and `serial_putchar` RPC calls handle basic serial communication between the client and the server. Both involve simple interactions, where the server processes a single character and responds appropriately.

Serial Get Character: In the `serial_getchar` RPC call, the client requests a character from the server. The server reads the character from the console and responds with it. The client message consists of a message type (`AOS_RPC_SERIAL_GETCHAR`), while the server message includes an acknowledgement type (`AOS_RPC_SERIAL_GETCHAR_ACK`) and the character read from the console.

Serial Put Character: In the `serial_putchar` RPC call, the client sends a character to the server. The server prints the character to the console and responds with an acknowledgement. The client message consists of a message type (`AOS_RPC_SERIAL_PUTCHAR`) and the character to be printed. The server message contains an acknowledgement type (`AOS_RPC_SERIAL_PUTCHAR_ACK`).

6.8 RAM Request

In the `ram_request` RPC call, the client requests a RAM capability from the server, specifying the minimum size and alignment requirements. The server allocates the requested RAM, sends the corresponding capability back to the client, and includes the size of the allocated memory in the acknowledgement.

The client message consists of the following fields: a message type (`AOS_RPC_MEMORY_REQUEST`), the requested size in bytes, and the required alignment.

The server message consists of the following fields: an acknowledgement type (`AOS_RPC_MEMORY_REQUEST_ACK`), the allocated capability, and the size of the allocated memory in bytes.

6.9 Spawn With Command Line/Default Arguments

Both functions enable a client to request the spawning of a new process, specifying either a full command line or default arguments.

Spawn with Command Line: The Spawn with Command Line RPC call allows the client to request a new process to be spawned using a specified command line, including arguments. The client message includes the message type (`AOS_RPC_PROCESS_SPAWN_CMDLINE`), the length of the command line string, and up to six words of packed characters representing the command line. The server, upon receiving the message, unpacks the command line, spawns the process, and responds with an acknowledgement message containing the PID of the newly spawned process.

Spawn with Default Arguments: The Spawn with Default Arguments RPC call offers a simpler way to spawn a new process using only the binary's path and default arguments. The client message includes the message type (`AOS_RPC_PROCESS_SPAWN_ARGS`), the length of the binary path string, and up to six words of packed characters representing the path. The server unpacks the path, spawns the process with default arguments, and sends an acknowledgement message with the PID of the spawned process.

6.10 Spawn With Capabilities

In the `spawn_with_caps` RPC call, we encountered the challenge of passing multiple capabilities from the child process to the `init` process using an LMP channel, which can only pass a single capability at a time. To resolve this, the child process creates a dedicated L2 CNode, copies the desired capabilities into that CNode, and sends the CNode's capability over the LMP channel.

On the server side, the `init` process unpacks the received capability, allocates a slot in its **L1 CNode** within its CSpace, and copies the L2 CNode's capabilities into that slot. It then reconstructs the array of capabilities and spawns the process using the reconstructed array.

The client message includes the following fields: a message type (`AOS_RPC_PROCESS_SPAWN_CAPS`), the number of arguments, up to six words of payload for the arguments, and the capability for the dedicated L2 CNode.

The server message consists of an acknowledgement type (`AOS_RPC_PROCESS_SPAWN_CAPS_ACK`) and the PID of the newly spawned child process.

6.11 Process Management

Process management functions with RPC follow a consistent pattern: the `init` process serves as the process manager. When a child process needs process management information, it invokes the appropriate client-side process management function, which sends a message with a message type `AOS_RPC_PROCESS_GET_*`. The message payload contains all relevant details, such as the name or PID of the process being queried.

Upon receiving the message, the `init` process calls the corresponding process management functions implemented in the previous milestone to retrieve the requested information. It then sends the collected information back to the client using a message type `AOS_RPC_PROCESS_GET_*_ACK`. On the client side, the receive handler unpacks this message and returns the requested information to the caller.

6.12 Suspend and Resume

Suspend: The `suspend` RPC call pauses a process by sending its PID along with the message type (`AOS_RPC_SUSPEND`) to the `init` process. Upon receiving the request, the `init` process halts the target process by removing its dispatcher from the scheduler and responds with an acknowledgement (`AOS_RPC_SUSPEND_ACK`).

Resume: The `resume` RPC call restarts a previously paused process by providing its PID and the message type (`AOS_RPC_RESUME`). The `init` process reinserts the dispatcher into the scheduler, allowing the process to resume, and sends an acknowledgement (`AOS_RPC_RESUME_ACK`) to the client.

6.13 Wait

To implement the wait functionality, we extend the process management structure by attaching a waiting process list to each process management node. When process `A` attempts to wait on process `B`, it registers itself with the process manager by adding a node to the waiting process list associated with process `B`'s management node. This wait node contains process `A`'s LMP channel, enabling the process manager to trigger process `A` in the future by sending a message to this dedicated LMP channel (see Section 6.13).

As with all other RPC function calls, the waiting process sends a message to the `init` process and then waits for a response on the default waitset. However, in this case, the `init` process registers the waiting process without immediately sending a response. Instead, the child process effectively waits by remaining blocked on the response message, allowing the wait functionality to be seamlessly integrated into the existing RPC mechanism.

The client message includes the following fields: a message type (`AOS_RPC_WAIT`), the PID of the process that is waited, and the PID of the process that is waiting.

The server message consists of an acknowledgement type (`AOS_RPC_WAIT_ACK`) and the exit status of the process that has been waited.

6.14 Exit, Kill and Kill All

The exit, kill, and kill all functions handle scenarios where a process terminates or one process terminates another process or multiple processes. In all cases, the `init` process as the process manager stops the execution of the target process(es) by removing its dispatcher from the scheduler.

Exit: The exit RPC call is used to notify the process manager when a process terminates gracefully. Unlike other RPC calls, this function does not involve waiting for a response, as the terminating process can no longer handle it. The exit RPC is automatically triggered when a process ends, sending a message containing the message type (`AOS_RPC_EXIT`), the PID of the terminating process, and its exit status to the process manager.

Kill and Kill All: In the kill and kill all RPC calls, the client sends the PID or name of the target process(es) along with the appropriate message type (`AOS_RPC_KILL` or (`AOS_RPC_KILL_ALL`)). The `init` process locates the

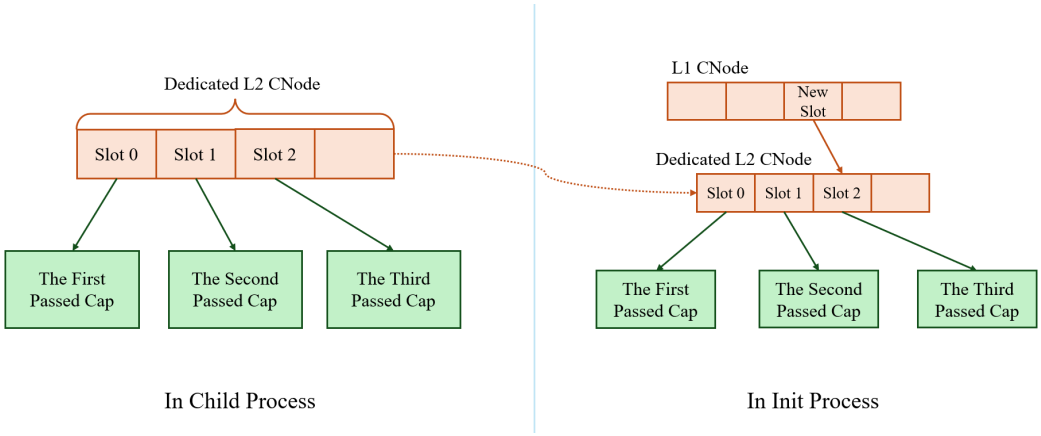

Figure 11: Passing Capabilities in RPC Spawn With Capabilities

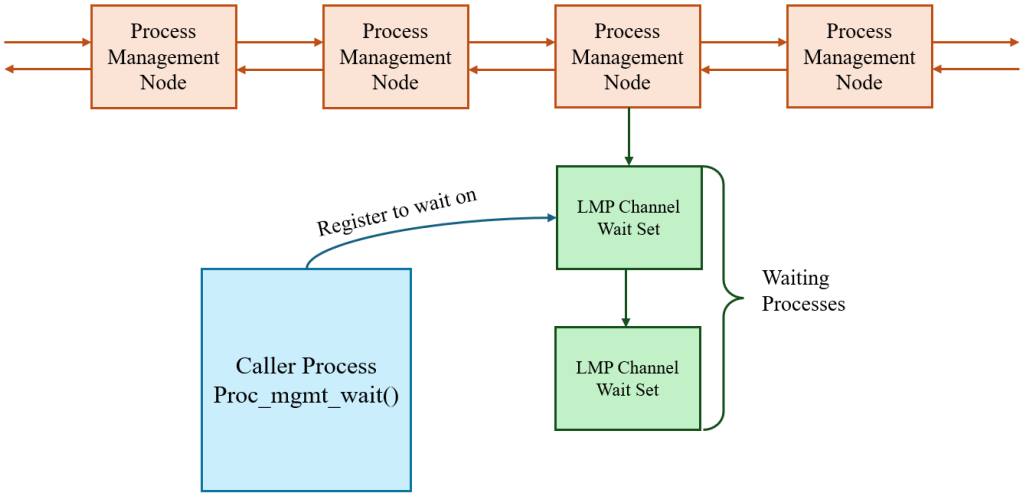

Figure 12: Extended Process Management Structure for Waiting

specified process(es), terminates them, and responds with the appropriate acknowledgement message (`AOS_RPC_KILL_ACK` or (`AOS_RPC_KILL_ALL_ACK`)).

Once a process is terminated, the process manager traverses the waiting processes attached to the terminated process. It sends a message with the type `AOS_RPC_WAIT_ACK` to the recorded LMP channels of the waiting processes, waking them up. This allows the waiting processes to receive the expected message and resume execution (also see Section 6.12).

6.15 Thread Safety

The issue of multithreading arose toward the end of the development process when we began testing under a multithreaded context, where multiple threads attempted to send messages to the `init` process simultaneously. This uncovered problems we had not previously considered.

6.15.1 Problems

In the current workflow, any thread that wants to make an RPC call to the `init` process must register a dedicated receive handler, send the message, and wait for a response. Since the response may not arrive immediately, a call to `event_dispatch` is made, which waits for the response on a waitset and yields the thread until the response is received. Up to this point, everything works as expected. However, if a second thread attempts to make an RPC call while the first thread is still waiting for a response, the process fails.

The failure occurs because the response for the first thread has not yet arrived, and thus the receive handler on the pro-

cess's LMP channel remains registered. When the second thread attempts to register a new receive handler on the same channel, it results in an error since a channel can only have one registered handler at a time, and all threads in a process share the same channel.

6.15.2 Solution

To address this issue, we implemented a solution using a conditional variable stored in the `aos_rpc` structure. At the beginning of any RPC function call, a thread attempting to register a dedicated receive handler must first wait on the conditional variable. If the process's LMP channel already has a receive handler registered, the thread cannot proceed; it must wait until the channel is free.

The thread that holds the conditional variable performs the RPC call and waits for a response. While other threads wait for the channel to become available, the register state remains locked, preventing additional RPC calls from being attempted. When the response arrives, the registered receive handler is triggered and subsequently deregistered. At this point, the conditional variable signals the next waiting thread, allowing it to proceed and register its receive handler.

This approach ensures thread safety by enforcing that only one thread at a time can use the LMP channel to communicate with the `init` process, effectively resolving the multithreading issue.

6.16 Interaction With Other Components

The RPC system built on top of the LMP channel interacts with several key components of the system, including:

1. MM (Memory Manager) and `malloc`
   The memory manager and `malloc` are essential to the implementation of RPC. For instance, the RAM request function relies on the memory manager to allocate a specified amount of memory. Additionally, `malloc` is widely used across the system to allocate memory chunks as needed for various operations.

2. Spawning
   The RPC mechanism depends on the creation of an LMP channel during the process spawning phase. As part of the spawning process, the endpoint capability for the `init` process is stored in the newly spawned child's CSpace, enabling communication between the two processes.

3. Process Management
   To support process management through RPC function calls, the `init` process acts as the process manager. When it receives process management requests, it collects the necessary information and responds to the client process.

6.17 Limitations

Unlike previous milestones, where system designs had clear strengths and weaknesses, the limitations of our current implementation of RPC over LMP primarily stem from a lack of structured design.

Overall, the design of RPC over LMP is raw, flat, and lacks clear abstractions or layered separation. The messaging layer, responsible for marshaling messages, and the transport layer, responsible for sending messages, are heavily conflated. This blending of responsibilities makes the system difficult to extend or adapt. For instance, if we were to replace the transport layer or switch to a different communication mechanism, such as a different channel type or shared memory, the entire implementation would require substantial changes, since these layers are not independent. A well-designed system would decouple these layers, enabling cleaner interfaces and greater flexibility for extending or upgrading the system without disrupting existing functionality. Addressing this design flaw could significantly improve the modularity, maintainability, and scalability of the RPC system.

Additionally, using a single channel between the `init` process and each child process can overwhelm the `init` process with diverse types of requests, creating a bottleneck. In the future, a more scalable approach could involve setting up dedicated processes, such as a memory server, serial server, and process server, to handle specific types of requests. This would distribute the workload, reduce the burden on the `init` process, and improve overall system performance.

6.18 Retrospective and Improvements

This milestone was a particularly chaotic time for our team, but it provided invaluable insights into the balance between design and implementation. While diving directly into implementation can seem appealing due to the immediate results it yields, neglecting proper design and abstraction creates significant long-term challenges. Without thoughtful design, problems are often oversimplified, leading to systems that are difficult to extend, maintain, and manage. This milestone highlighted the importance of allocating sufficient time for planning and designing robust abstractions before jumping into implementation, ensuring the system is prepared to handle future complexities.

Beyond technical challenges, this milestone also deepened our understanding of team management and dynamics, particularly as the term progressed and workloads increased.

Coordinating schedules, managing task dependencies, and maintaining effective communication became more challenging. By navigating these challenges, we improved our ability to distribute workload effectively and maintain productivity under pressure.

The most impactful technical takeaway from this milestone is the need to introduce a layer of abstraction to separate the messaging layer and the transport layer. By doing so, we could improve modularity, making the system more flexible and easier to extend. For instance, with a well-defined messaging layer, replacing the transport layer or adding new communication mechanisms would require minimal changes. This separation would also streamline debugging and testing, as each layer could be evaluated independently. Implementing such an abstraction in future work would greatly enhance the maintainability and scalability of the system.

7 Milestone 5

The milestone 5 can be divided into two parts: booting a second core and setting up initial inter-core communication. The first part is considered similar to milestone 3, which requires precise and correct implementation of procedures. The second part, inter-core communication, leaves significant freedom in the implementation choices, and the design decisions are explained below.

7.1 Booting a Second Core

7.1.1 Preparing Data Structures

The process starts by creating data structures to boot a core. There are four data structures that need to be created: `armv8_core_data`, `kernel stack`, `URPC frame`, and `kcb`. Although the book suggests it is possible to create a big frame containing the first three data structures, separate frames were chosen for easier debugging. For `kcb` only, it was retyped to type `ObjType_KernelControlBlock` as instructed in the book.

7.1.2 Preparing ELFs

The ELF loading process is:

1. Locating the ELF image's capability through its path from the `bootinfo`
2. Loading the ELF image into memory
3. Finding the ELF Entry Point
4. Loading the ELF into memory
5. Relocating the ELF

There are three differences from milestone 3's ELF loading procedure:

1. **Finding Entry Point**: The entry point of the ELF image needs to be manually found by supplying the entry function's name.
2. **ELF Loading**: The ELF loading process is simpler than milestone 3 as there is only one mappable region inside each ELF image. This time only a simple function call to 'load_elf_binary' is needed without requiring a callback function.
3. **ELF Relocation**: The kernel driver ELF is relocated after loading, as it will run in the kernel virtual address space, instead of the normal address space it was compiled for.

The above-described techniques are used to load `boot driver`, `cpu driver`, and `init`.

7.1.3 Preparing Second Core's Memory

In this part, the memory for core 1 is statically allocated, using core 0's `ram_alloc`.

It is acknowledged that there are other better designs, such as allowing the second core to request more memory on demand. However, for simplicity in this milestone, the simpler approach was chosen. Initially, not enough memory was allocated, and after the core booted, memory errors were encountered. The second core's memory was eventually finalized to be `16M`.

7.1.4 URPC Frame Structure and Setup

The URPC frame is divided into two page-sized sections: one reserved for core 0 and the other reserved for core 1. This data structure will be further explained in milestone 6.

Here are three frames that need to be passed to the second core using the URPC frame, and they are copied into core 0's part of the URPC frame, for core 1 to read and forge capabilities. Since directly sending capabilities between cores is not an option, the physical address and size of the frame are sent instead, and it is the second core's responsibility to forge the capability.

1. **Second Core's Memory**: Physical address and size of the memory reserved for the second core.
2. **bootinfo**: A copy of the current core's `bootinfo`.
3. **mmstring**: A copy of the current core's `mmstring`. This is required to find a program by name from `bootinfo`.

7.1.5 Setting Up core_data

Following the instructions, the following fields are set up in `core_data`:

1. boot magic
2. CPU driver stack and entry point
3. command line arguments
4. second core's memory and size
5. URPC frame's physical address and size
6. newly created `init`'s base and size
7. source and destination core IDs

7.1.6 Memory Barrier and Cache Invalidating

In order for the second core to correctly read the data, cache invalidations and memory barriers to overcome ARM's weak memory model are necessary. In the design, a set of memory barriers are called before and after cache invalidations. Each set of memory barriers includes `dsb sy`, `dmb sy`, and `isb`.

Cache invalidations are performed on every data structure that could be shared between cores as listed below. Better safe than sorry!

1. boot driver
2. CPU driver
3. kernel stack
4. second core's memory
5. `kcb`
6. `init`/monitor
7. core_data
8. second core's `bootinfo`

7.1.7 Spawning the Second Core

After setting everything up, `invoke_monitor_spawn_core` is called to spawn the second core.

7.1.8 Waiting for Core 1's Response

As later the UMP channel needs to be set up after core 1 boots, core 0 needs to wait for core 1 to boot up. A simple polling mechanism is used to wait for core 1 to write a number 1 into its UMP buffer.

7.2 Second Core's Self-Setup

This section describes the second core's self-setup process inside its `init` (`app_main`) after it boots up.

7.2.1 Forging Capabilities

As passing capabilities between cores is not an option, capabilities need to be forged from the physical addresses and sizes passed from core 0. The following capabilities are forged in the second core's `init`:

1. forging `bootinfo` capability into `bootinfo`'s known cslot.
2. mapping `bootinfo` into virtual memory and assigning global variable `bi` to it.
3. forging second core's ram and calling `mm_add` to add it to the memory manager.
4. creating a cnode at `mmstring`'s known cslot and copying the passed `mmstring` content into it.
5. forging capabilities of modules in `bootinfo` into cnode `cnode_module`. Because at this stage there is access to the entire bootinfo, that will give the size and physical addresses of all the modules, and they can be forged in a loop.

7.2.2 Signal Core 0 Core Has Booted

At this stage, core 0's `coreboot` function should be in a polling loop waiting for core 1 to write data into its own UMP buffer. Therefore, a number 1 is written into the beginning of core 1's UMP buffer.

7.3 UMP Polling

This section discusses the temporary solution in milestone 5 to receive messages from UMP polling - using a dedicated polling process. First, the problem is described, followed by the motivation, and the approach (for milestone 5 only).

7.3.1 Problem

In milestone 5, when a basic form of inter-core communication was implemented - allowing the first core to signal the second core to spawn a process. The intuition was that the second core would be in an infinite loop reading the UMP frame, and upon receiving the data written from the first core, it would spawn a process. The initial polling location selected was the second core's `init` process, and the message was indeed received.

However, it was soon realized that the polling was blocking the local LMP IPC because its waitset was being blocked from dispatching in `init`.

7.3.2 Solution

The next solution considered was to create another thread-/process to do the polling. Eventually, a process was chosen because there is more familiarity with processes than threads.

In the polling process, it reads the UMP frame, which is in a pre-defined cnode, in an infinite loop. When it receives a message, it will use local LMP to spawn a process. After spawning, it clears the message and re-enters the loop.

In milestone 6, this will be replaced by UMP polling channel waitset.

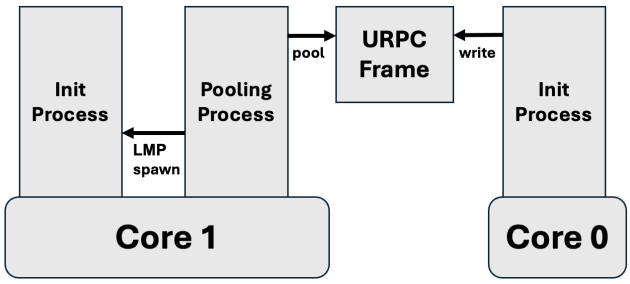

Figure 13: URPC Polling for M5

7.4 Interaction With Other Components

The core booting process relies on various components of the overall system, including:

1. Memory Manager (MM)
   After booting up, core 1 requires memory to operate and a memory manager to allocate physical memory. In our current design, the memory manager on core 1 can allocate and manage its statically allocated memory resources, similar to the memory manager in core 0.

2. Paging System
   The paging system plays a critical role during the core booting process. By mapping the core data structures into core 0's address space, core 0 can pass the booting information needed by the child core.

7.5 Limitations

7.5.1 URPC Polling

Using a dedicated process for polling is a temporary solution rather than a proper one. It is slow and lacks a unified way to manage message queues. Because only core 1 is polling and not core 0, unless core 0 explicitly reads messages from core 1, it will not be able to know the message. The current solution is just a bare minimum to have some inter-core communication and is not expandable. In milestone 6, a more sophisticated solution—UMP channel waitset—will be implemented to handle the messages.

7.5.2 Spawning Core Failure Handling

As described above, the current implementation relies on a polling loop for core 0 to determine if core 1 has booted. This design works fine when core 1 boots successfully, but if core 1 fails to boot, core 0 will be stuck in the polling loop forever. A better design would include a timeout mechanism to handle this situation.

7.5.3 Memory Allocation

The second core's memory is statically allocated. A better design would allow the second core to request more memory on demand using URPC from core 0. However, this design is still not perfect because it separates memory management between cores, meaning a region of memory cannot be available to more than one core. A better design would involve a centralized memory manager collaborating with all cores.

7.6 Retrospective and Improvements

This milestone provided the group with great insight into the structure and design of Barrelfish because it required digging into resources that are already available on core 0, in order to set up core 1. For example, when initially trying to spawn a process in core 1's `init`, it was impossible to get the module from the `bootinfo` by providing its name. Later, it was found that this was due to the `mmstring` not being present in core 1, which is a field automatically set up in core 0. Although this required some digging, the group now understands how the system finds a program's `memregion` just by its path, which was a black box in milestone 3.

The group also made a great design decision in this milestone: separating the URPC frame into two parts, one for core 0 and one for core 1. Although this design is not discussed yet (as it is a part of milestone 6), it was proposed in this milestone and was a great design decision.

8 Milestone 6

In this milestone, we refined and improved our inter-core communication by implementing User Level Message Passing (UMP). Compared to previous milestones, during this milestone we had the most freedom to experiment and create our own designs with it's own benefits and trade-offs. Additionally, this milestone made us go back and change several aspects of our design, in order to better accommodate communicating between cores.

8.1 Data Structures

8.1.1 Shared Frame

As explained in Milestone 5, we used the shared URPC frame to communicate between the cores. This frame is divided in half, with the first half reversed for communication to/from the parent core (core 0), and the second being used for the child core (core 1). The send frame for one core is the receive frame for the other, and vice versa.

Each divided half of the frame is further broken down into the following sections:

1. **Head pointer** - The current head pointer of the ring buffer (64 bits)
2. **Tail pointer** - The current tail pointer of the ring buffer (64 bits)
3. **Ring Buffer** - The actual ring buffer, holding up to 8 messages.

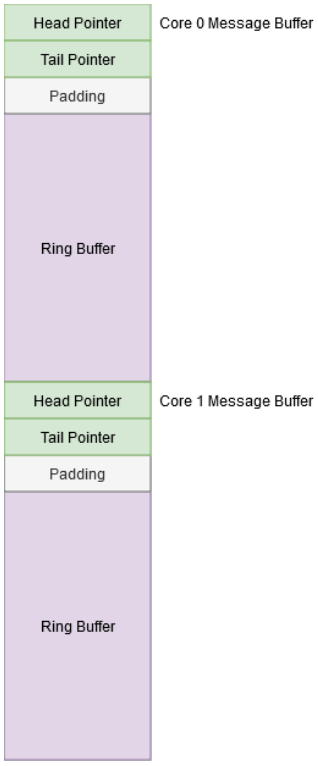

Figure 14: Shared Frame

8.1.2 Messages

Messages for UMP were designed to match as closely as possible to the messages sent using LMP. This was a design decision to enable us to easily translate between LMP and UMP requests with minimal changes.

Our UMP messages are defined as 64 bytes, divided into 8 words, of 8 bytes in length.

Additionally, our messages are sent using the following definition:

1. **Header** - The 4 most significant bytes are reserved for the PID of the process that is sending the message, the 4 least significant bytes are reserved for the message type. The message type definition is shared with LMP (see Milestone 4).
2. **Capability Base Address (optional)** - If ram/a frame is being sent, this message will be the base address
3. **Capability Size** - If ram/a frame is being sent, this message will be the size of the capability.
4. The rest of the message has no official standard

The header requiring the PID of the process that sent the message is done so that we can easily tell from an incoming message over the UMP channel which process had initially sent it, in order to easily route the responses back.

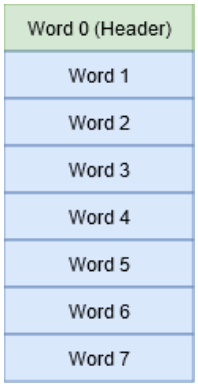

Figure 15: UMP Message Format

8.2 API

To send UMP messages, we defined a separate API outside of `AOS_RPC`, exclusively for sending messages over UMP. Since the mechanisms required for creating, sending, and receiving UMP messages are unrelated to LMP, and so this decision was made to reduce code duplication, in addition for allowing the user to explicitly send a UMP message without having to go through the `AOS_RPC` API.

This API was created to match as closely as possible to that of `AOS_RPC`, both because of inspiration, and to keep API calls consistent for the end user. We defined API's for:

1. Initializing UMP connection
2. UMP Handshake
3. Receiving UMP messages
4. UMP message routing
5. Sending UMP messages
6. Forwarding the UMP message to a process using LMP

Additionally, there are functions easily sending specific UMP messages, such as:

1. Spawning with command line / default arguments
2. Suspending, Resuming, Waiting, and Killing processes
3. Getting Meta-data, like status, exit code, and PID of processes
4. Sending numbers
5. Requesting Ram

The complete list of API functions can be found in the appendix.

8.3 Changes from Previous Milestones

During this milestone, there were many changes made to older milestones in order to accommodate sending messages between different cores. These were required because of a need to be able to find which core a message originated from, and which process on the other core sent the message.

Due to this, there were changes to the allocation of PID's, and additional meta-data was added to the LMP message headers.

8.3.1 PID's

Given each process on a core has a unique PID, PID's were decided as a good way to track which process a message request came from. Previously in Milestone 3, PID's were assigned using an incrementing counter stored in a global variable. While this worked at the time, this would mean that all PID's within a core were unique, but there may be duplicate PID's across cores.

In order to make the PID a useful identifier for processes across cores, the PID was changed to also contain meta-data on which core it's corresponding process was running on. Thus, the most significant bit of the PID was changed to indicate which core the process was running on, being 0 for core 0, and 1 for core 1. A single bit is enough for our purposes, since our implementation only has 2 cores to work with.

Thus, with the new PID's, we can determine which core a PID corresponds to by looking at the most significant bit.

8.3.2 LMP Messages

Previously during Milestone 4, the first word (the header) of a message contained the enum type for what message it was for, ex. `AOS_RPC_HANDSHAKE`. This worked great for intra-core communcation between processes, since the `init` process has access to all endpoint capabilities to the LMP channels for the processes running on the same core. During milestone 6, we included the additional meta-data of the sending process PID as part of the header. This was done to keep messages on LMP and UMP to be as similar as possible, so that translation between the two protocols could be simpler. Adding this meta-data also made it easier to track which process had sent certain messages, which was useful (as later explained in algorithms), to forward messages received by `init` over UMP to be sent back to the corresponding process.

8.3.3 UMP Server

In milestone 5, we had an initial implementation of a UMP server, similar to `init`, with an instance running on each core. During this milestone, we decided to abandon this approach in favour of waitsets, which is further explored later in this report.

8.3.4 Initializing UMP Communication + Handshake

UMP initialization stays mostly unchanged compared to milestone 5. The key differences are as follows:

1. Messages now follow the new message definition, where the initial message contains all the information necessary for the new core to boot
2. After successfully calling `invoke_monitor_spawn_core`, core 0 will register a polled-waitset on the URPC frame to receive messages from core 1
3. Once the core 1 boots, it will establish a polled-waitset to receive messages from core 0
4. Once the core 1 boots, it will send and ack to core 0

8.4 Overall Design

Our approach to intra-core communication builds upon our original design, where `init` functions as a central server. Similar to our implementation of RPC over LMP, child processes act as clients, using the centralized `init` server for communication. Now, we have extended `init`'s role to

handle inter-core communication via UMP - `init` now routes messages from child processes for the other core to the other core through the UMP channel, and routes responses from the other core back to the sending child process.

The decision to route all UMP messages through `init` to be forwarded was a pragmatic one. Firstly, our LMP messages from milestone 4 was implemented using a single channel to `init`, and so going with this approach let us build on top of our current design, rather than building an entirely separate system. Secondly, there is only a single `init` process, and one URPC frame - coupling the two concepts together alleviated some of the mental overhead of reasoning messages. Finally, having only `init` read and write to the shared buffer helps reduce concurrency issues, since there is only one process to account of performing all the actions.

At a high level, our inter-core communication works as follows:

1. A child process wants to send a message to the other core. It will compose this message, and send it to it's `init` process running on it's own core
2. The `init` process receives the message from the child process and parses it
3. If the message is destined for the other core, it will forward it to the other core over the UMP channel
4. The `init` process on the other core receives the message and parses it
5. Based on the message, it will perform some actions, compose a response, and send the response back over the message buffer
6. The `init` process receives the message from the other core, parses it
7. The `init` process determines the message as a response, and forwards it to the waiting child process

The following subsections will describe the lower-level details of the implementation

8.4.1 Composing Messages

Typically, if UMP is invoked through the `AOS_RPC` API, it will need to translate the input LMP message into a UMP message. Given we have taken the time to make the API's as similar as possible - this is trivial. We simply keep the same header from the LMP message, and then transfer the contents of the message into the UMP message, ignoring the endpoint capability. Typically this results in a near 1-1 mapping, where the first word of a LMP message will be exactly the same as the first word of a UMP message.

Transferring Capabilities

The one exception to the easy translation is with transferring RAM and frame capabilities, which are required for RAM requests, and large messages. In LMP, this is achieved by sending the `capref` in addition to the size of the RAM/Frame region, and then allocating it into the receiving CSpace. In UMP, this isn't possible, because `capref`'s are not valid across cores.

In order to overcome this, we instead send the capability's physical base address, as well as the size of that capability as part of the message. When this is done, we assume that the other core now has total ownership of that resource.

8.4.2 Sending Messages/Responses

Sending Messages When `init` receives a message, it needs to be able to determine if the message is intra-core, or inter-core. The process of routing sent messages depends on both the contents of the message, as well as the headers.

When sending a message, typically most of the `AOS_RPC` API's provide a mechanism that allows us to easily determine which core it is for. For example the miscellaneous `spawn_with_*` commands have the core number as a parameter - if the core specified is not our core, we simply forward the request to the other core. Other messages, like `proc_mgmt_get_name`, have a PID as an input. Since we had changed the PID to have it's most significant bit represent the core that it is running on, if we find that the PID corresonds to the process on the other core, we also forward the response to the other core.

This is also true for the process management functions within `proc_mgmt`, if an invocation specifies a different core, it will forward that request to the other core via UMP.

For requests that both do not have a either a core or PID argument, our policies are that these commands must be issued using our UMP endpoints directly, which are specified in the Appendix. This design decision was made because our design would not be able to determine which of either LMP or UMP would be required, since there is no specificity of which core the user wants to send the request to.

Tracking Sent Messages

Once we send a message, we store meta-data in a doubly linked-list in order to properly forward the response back once it is received. In particular, the list stores:

1. The PID of the sending process - This is stored so we can associate the response back to this process

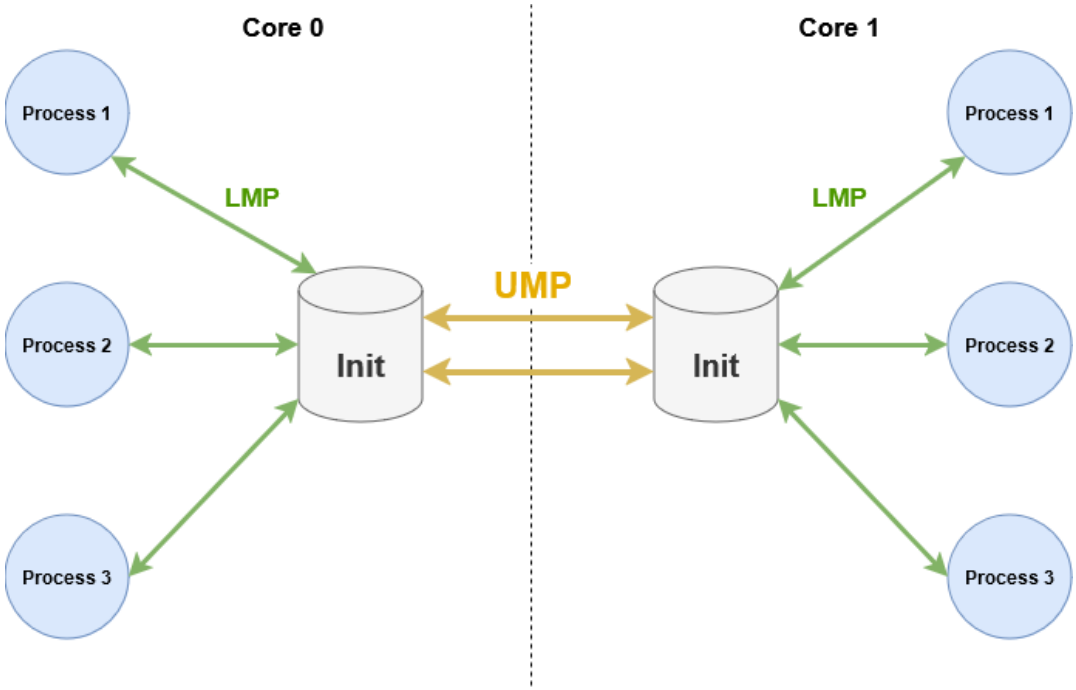

Figure 16: UMP Message passing

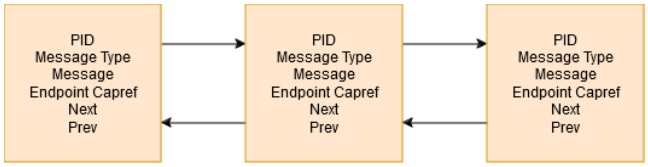

Figure 17: UMP Response Linked List

2. The message type of the sent message - This is stored so that if a process sent multiple messages, we can tell them apart

3. The actual message that was sent - This is stored so that once the response is received, we can return values if pointers were provided

4. The endpoint capability of this process - This is stored so that we can forward the response

When the node is added, it is simply added to the front of the list. This was because there is no guarantees in what order responses will come in, and so there is no way to optimize the list for parsing responses, and thus we optimize the runtime of adding to $O(1)$.

Road not taken - Alternative Data Structures

A linked list data structure was chosen mostly for it's simplicity. Since a single process can send multiple messages over UMP, and send messages of the same type, it limits the potential optimizations that could be implemented.

An alternative data structure could be a hashmap. A hashmap is appealing because every process already has a unique "key" identifier, being the PID. Additionally, this idea was attractive because hashmaps have O(1) lookup times, which in principle could lead to performance increases. However, this was not pursued further upon realization that a single process could send multiple messages, and multiple messages of the same type. This would mean that we would likely need the value of the hashmap field to be the head of a linked list, in order to track individual messages sent by the process.

This would increase complexity immensely, since we would have all the complexity of a linked list, in conjunction to all the complexity of a hashmap. Finally, our message buffer was only able to hold up to 8 messages in flight at a given time, meaning that our data structure would only need to contain up to 8 elements. With this consideration, the hashmap would likely only decrease performance, since it would bring along much more overhead than just traversing a linked list.

Sending Responses

When `init` is routing a response instead of a message, it will parse the header to get it's PID. If the PID is originates from the other core, we forward the message over the UMP

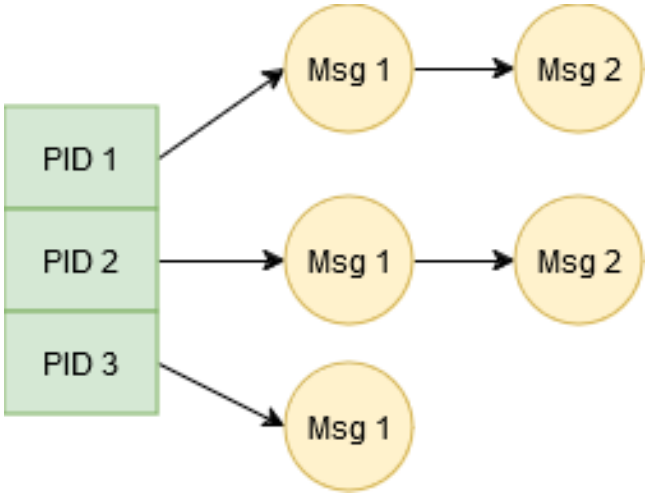

Figure 18: Alternative Hashmap

channel. When we do this, we do not expect a response back, and do not track any meta-data.

8.4.3 Sending Messages - Writing to the Ring Buffer

As previously explained during Milestone 5, the shared URPC frame is used to hold two ring buffers to sending and receiving messages. The ring buffer is an efficient data structure for inter-channel communication since it maintains the semantics of a FIFO message buffer, but is well designed for a fixed-size memory region.

The design follows a classic producer-consumer model, with the sender (producer) writing data into the buffer, and the receiver (consumer) reading data from it. Communication proceeds through careful use of the head pointer and tail pointer, which are manipulated in a circular manner to maintain the FIFO order.

When a `init` wants to send a message, it writes to the memory position pointed to by the head pointer, in the other core's receive ring buffer. However, before writing, it ensures there is enough available space by checking if the head pointer would collide with the tail pointer, preventing message overwriting. Once the message is written, the head pointer advances, wrapping back to the beginning of the buffer if necessary.

Conversely, when `init` wants to read a message, it checks whether there are unread messages by comparing the head pointer with the tail pointer. If messages are available, `init` reads the data at the position indicated by the tail pointer and then increments the tail pointer in a circular manner, until the buffer is considered empty.

Currently in our design, if the ring buffer is full, and `init` attempts to send additional messages, it will result in an error, since we there is not enough memory to send the message.

8.4.4 Receiving Messages - Reading from the Ring Buffer with Waitsets

Each core has a polled-waitset registered on it's receive buffer. This waitset continually polls on the ring buffer, creating an interrupt when the buffer is no longer empty. This message is then read and routed, which increments the tail pointer, and the waitset is re-registered onto the receive buffer.

8.4.5 Road Not Taken - Monitor Process

Initially, our design during Milestone 5 had a separate process on each core dedicated to receiving messages. This decision was made to mimic the "monitor" process that exists in barrelfish upstream to coordinate messages between the cores. This process would continually poll the receive buffer, and when a message arrived, it would send a LMP message to `init` to be routed and managed.

While this designed worked for the simple implementation in milestone 5, it quickly became apparent that this solution was not optimal, due to the additional complexity of managing and launching a separate process.

1. This solution introduced potential race conditions, where the buffer could fill up because the process hadn't been launched yet
2. Lead to potential user error, where multiple "monitor's" could potentially be created - or additional work required to ensure this couldn't happen
3. Unfamiliar API - since we would be designing this from scratch with no framework, the API that was established to use was unintuitive and did not match the rest of the message passing algorithms

Thus, we decided to use polled waitsets, in order to avoid continually polling the channel to see if messages have arrived.

8.4.6 Routing Received Messages

After the message is received, we will need to figure out what to do with it.

Commands/Requests

For these kinds of messages, `init` will execute the command/request that the message asked for. Once this is completed, The response is created, and sent back over the

UMP channel. See sending responses for more details.

Responses
When it is determined that a response was received, `init` will look up in the linked list data structure for the node that corresponds to this response using the PID from the response's header, and the type of response the message is for. If the node is found, the response values are populated if there are any, and then the response is sent back to the other process using the stored endpoint capability. Once the response is sent back, the corresponding node in the linked list is removed.

If the response node is not found, we will instead return an error.

8.5 Interaction With Other Components

The RPC system built on top of the UMP channel interacts with different components in the system:

1. RPC over LMP
   As introduced in the current milestone, anytime a child process in a core wants to interact with another core, it first routes its message to its local `init` by using RPC over LMP and waiting on its LMP channel. Similarly, once the response is returned, the local `init` parses it and sends the message to the registered child process through LMP channel.

2. Paging System
   RPC over UMP interacts with the paging system in multiple ways. The URPC frame is mapped to `init`'s address space using the paging system, so it can be read and written by the `init` process during RPC over UMP.

3. Process Management
   RPC over UMP facilitates cross-core process management by actively gathering process management information from both cores. This involves merging the information collected by two `init` processes, which serves as the process managers on both the parent core and the child core.

8.6 Limitations

The limitations of our design partially come from the same limitations of our system from milestone 4.

Much like milestone 4, RPC over UMP is raw, flat, and lacks clear abstractions or layered separation. The messaging layer, responsible for marshaling messages, and the transport layer, responsible for sending messages, are heavily conflated. Thus, we had to develop an entirely alternative system in order to extend the functionality over a separate channel. This difference may be confusing to some users, where certain `AOS_RPC` commands are able to work over UMP, and others require special handling.

Additionally, the API's that were developed much more bare bones than those that were provided for LMP. While we did establish a protocol and standard, there is no enforcement on how these are composed to be sent, an no well defined process for parsing upon receiving - the end user needs to explicitly define a function to properly parse the messages sent over the channel.

Finally, in our current design, each buffer can only support up to 8 in-flight messages before filling up, and once the buffer is full, all attempts to write messages fill fail. This is unintuitive to the end user, and the end user of the UMP channel will need extra overhead to manage this resource limitation.

8.7 Retrospective and Improvements

In order to support more in-flight messages in the future, an additional data-structure could be created to act as a queue or buffer for messages that have yet to be written to the URPC frame. This would simplify the use of the protocol massively, as the consumers of the API would have the issue of a full buffer abstracted away. However, this has it's own challenges, since it would require additional overhead and mechanisms in order to detect when the buffer is full, and only then begin polling to write new messages as it clears.

In retrospect, if our milestone implementation was more fleshed out, and had more time had been spent creating a well defined message sending protocol, the implementation portion of this milestone could have gone much smoother. As discussed, there were many changes that needed to previous milestones in order to provide full UMP message passing functionality. This milestone highlighted the importance of the benefits of having a design be correct the first time, but also helped us learn to be flexible, and pivot our previous ideas to account for new requirements.

Beyond technical challenges, this milestone also deepened our understanding of team management and dynamics, particularly as this was the final milestone of the term, where the team had to balance many other deadlines, fixes and improvements to other parts of the system, and composing this report. Coordinating schedules, managing task dependencies, and maintaining effective communication became more challenging. By navigating these challenges, we improved our ability to distribute workload effectively and maintain productivity under pressure.

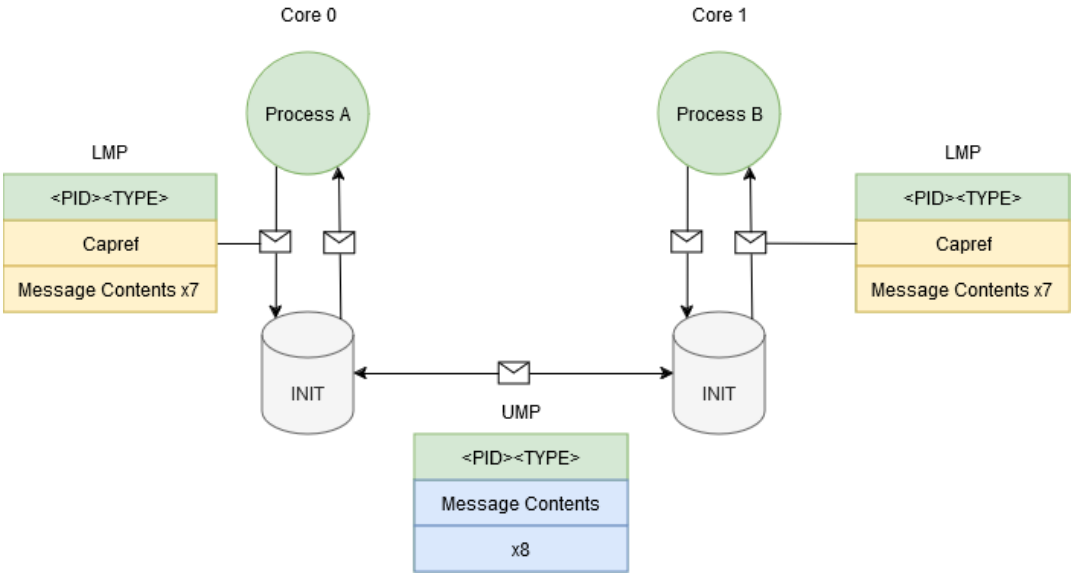

Figure 19: UMP Message passing Continued

9 Conclusion

Finally, we have reached the end of the semester, marking the conclusion of this project and the course. This journey has been both rewarding and challenging, blending moments of accomplishment with frustrations. It demanded much more than technical skills—it required us to address underspecified problems with creative solutions, collaborate effectively as a team, manage time under pressure, and tackle unexpected obstacles. Despite the difficulties, we have grown significantly, gaining invaluable lessons and skills along the way.

One of the most memorable technical challenges occurred during the development of the virtual memory management system in the first half of the course. Initially, we pursued a tree-like structure, which seemed promising but quickly became overly complex and unmanageable (see Section 3.1). Recognizing its flaws, we met as a team to explore alternative designs and ultimately adopted a simpler, more efficient structure. This experience emphasized the importance of evaluating design alternatives early and remaining flexible in our approach—an insight we aim to carry forward into future endeavours.

The second half of the project proved even more demanding than the first three milestones. As discussed in Section 6, the raw design of RPC over LMP created significant challenges, especially during the final milestone, where we integrated RPC over UMP. To enable efficient cross-core communication while reusing previously implemented RPC functions, we had to design new UMP APIs that seamlessly connected with our existing RPC calls. This effort was fraught with unexpected issues, such as difficulties extracting return values from receive handlers or timing out for earlier milestones' tests. These challenges underscored the critical importance of thoughtful design. Without careful planning during the design phase, the system became difficult to extend and maintain, leading to unnecessary chaos and inefficiency.

On the non-technical side, our team dynamics evolved as the semester progressed. While we grew closer and more cohesive, coordinating schedules and balancing workloads became increasingly difficult. To address these challenges, we encouraged open and early communication among team members. This proactive approach allowed us to identify bottlenecks, redistribute tasks to accommodate personal schedules, and maintain steady progress wherever possible. Fostering a collaborative and supportive environment was crucial in overcoming these hurdles and delivering our final project.

Looking back, this project has provided us with valuable transferable lessons and skills that we will continue to refine in the future:

1. **Invest time in design:** Developing systems of significant scale and complexity requires dedicating time to read, understand, and design. Far from being wasted, this effort is crucial for building robust, long-lasting systems.

2. **Be flexible and evaluate alternatives:** Exploring multiple design options, even through quick sketches or rough diagrams, can help identify potential pitfalls and prevent unnecessary complications. Flexibility in design choices ensures the system remains adaptable and efficient.

3. **Foster effective teamwork:** Successful collaboration extends beyond task division. It involves understanding task dependencies, proactively seeking and offering help, and maintaining consistent communication. Building trust and leveraging team members' strengths enable the group to adapt to challenges and achieve shared goals under tight deadlines.

There are countless more lessons we've learned along the way, too many to list here. As we conclude, we would like to thanks to the course team that guided and supported us throughout this journey. Thank you for helping us navigate this challenging but deeply rewarding experience.

9.1 Acknowledgement

This report was written with the assistance of GitHub Copilot and ChatGPT. It only fixes the grammar and sentence structure and does not create any content.

10 Appendix

10.1 Appendix A

```c
/**
 * \file
 * \brief UMP Channel for Barrelfish
 *
 * This header defines structures and functions for managing UMP (User Message Passing) channels
 * and their integration into Barrelfish's messaging system. It includes functions for initializing
 * UMP channels, sending and receiving messages, handling process management, and managing
 * responses to RPC calls.
 */

#ifndef _LIB_BARRELFISH_UMP_MESSAGES_H
#define _LIB_BARRELFISH_UMP_MESSAGES_H

#include <sys/cdefs.h>
#include <aos/waitset.h>
#include <assert.h>
#include <aos/aos_rpc.h>
#include <proc_mgmt/proc_mgmt.h>

/**
 * \brief Structure for initialization information shared between cores.
 */
struct init_struct {
    size_t      ram_size_bytes;          ///< Size of available RAM in bytes
    genvaddr_t  ram_paddr;               ///< Physical address of RAM
    size_t      bootinfo_frame_bytes;    ///< Size of bootinfo frame in bytes
    size_t      bootinfo_frame_paddr;    ///< Physical address of bootinfo frame
    size_t      mmstring_size;           ///< Size of the memory management string
    genvaddr_t  mmstring_paddr;          ///< Physical address of the memory management string
};

/**
 * \brief Structure representing a core message.
 */
struct core_msg {
    uint64_t arg[8];                     ///< Array to store arguments of the message
};

/**
 * \brief Circular buffer for storing core messages.
 */
struct core_msg_buf {
    uint64_t        head_ptr;            ///< Head pointer for reading messages
    uint64_t        tail_ptr;            ///< Tail pointer for writing messages
    uint64_t        padding[6];          ///< Padding for alignment
    struct core_msg buf[63];             ///< Array to store core messages
};

/**
 * \brief UMP channel structure.
 */
struct ump_chan {
```

```
    struct waitset_chanstate waitset_state;  ///< Waitset state for polling
    struct core_msg_buf     *ump_send_buf;   ///< Pointer to the UMP send buffer
    struct core_msg_buf     *ump_recv_buf;   ///< Pointer to the UMP receive buffer
    bool setup;                              ///< Flag indicating whether the channel is
        initialized
};

/**
 * \brief Response node for tracking asynchronous responses in UMP communication.
 */
struct ump_response_node {
    struct lmp_chan          *lc;        ///< Pointer to the LMP channel for the response
    domainid_t                pid;       ///< Process ID of the client
    enum aos_rpc_type         type;      ///< Type of the RPC response
    struct ump_response_node *next;      ///< Pointer to the next node in the list
    struct ump_response_node *prev;      ///< Pointer to the previous node in the list
    struct core_msg          *resp;      ///< Pointer to the response message
    bool                      done;      ///< Flag indicating whether the response is
        complete
};

/**
 * \brief Get the UMP channel.
 *
 * \returns A pointer to the current UMP channel.
 */
struct ump_chan *get_ump_chan(void);

/**
 * \brief Initialize a UMP channel.
 *
 * \param channel Pointer to the UMP channel structure.
 * \param base    Base address of the UMP shared memory.
 * \param core    Core ID to associate with the channel.
 *
 * \returns SYS_ERR_OK on success, or an error code on failure.
 */
errval_t ump_chan_init(struct ump_chan *channel, uintptr_t base, coreid_t core);

/**
 * \brief Register a receive handler for the UMP channel.
 *
 * \param channel Pointer to the UMP channel structure.
 * \param ws      Pointer to the waitset for event handling.
 * \param closure Closure to execute when a message is received.
 *
 * \returns SYS_ERR_OK on success, or an error code on failure.
 */
errval_t ump_chan_register_recv(struct ump_chan *channel, struct waitset *ws, struct
    event_closure closure);

/**
 * \brief Send a message over a UMP channel.
 *
 * \param buf Pointer to the UMP message buffer.
 * \param msg Pointer to the message to send.
 *
 * \returns SYS_ERR_OK on success, or an error code on failure.
 */
```

```
errval_t ump_send(struct core_msg_buf *buf, struct core_msg *msg);

/**
 * \brief Receive a message from a UMP channel.
 *
 * \param buf Pointer to the UMP message buffer.
 * \param msg Pointer to the structure to store the received message.
 *
 * \returns SYS_ERR_OK on success, or an error code on failure.
 */
errval_t ump_recv(struct core_msg_buf *buf, struct core_msg *msg);

/**
 * \brief Handle received messages on a UMP channel.
 *
 * \param arg Pointer to additional arguments for the handler.
 *
 * \returns SYS_ERR_OK on success, or an error code on failure.
 */
errval_t ump_recv_handler(void *arg);

/**
 * @brief Registers a callback for a specific response type and process ID.
 *
 * This function creates a new response node and links it to a global list of
 * response callbacks. It allows the system to handle specific responses by
 * matching the process ID and RPC type.
 *
 * @param[in] lc         Pointer to the LMP channel for sending responses.
 * @param[in] msg_pid    The process ID associated with the message.
 * @param[in] resp_type  The expected type of the response.
 *
 * @returns A pointer to the created response node on success.
 */
struct ump_response_node *ump_register_response_callback(struct lmp_chan *lc, domainid_t
    msg_pid,
                                                         enum aos_rpc_type resp_type);

/**
 * @brief Forwards a UMP response to the appropriate client based on the message type and
     process ID.
 *
 * This function searches the list of registered response callbacks for a match
 * with the given message type and process ID. If a match is found, it forwards
 * the response using LMP or marks the response as completed.
 *
 * @param[in] msg        Pointer to the UMP message to forward.
 * @param[in] msg_pid    The process ID associated with the message.
 * @param[in] resp_type  The type of the response to forward.
 */
void ump_forward_response(struct core_msg *msg, domainid_t msg_pid, enum aos_rpc_type
    resp_type);


errval_t ump_proc_spawn_with_cmdline(struct ump_chan *chan, const char *cmdline, coreid_t
    core,
                                     domainid_t *newpid);
errval_t ump_proc_spawn_with_default_args(struct ump_chan *chan, const char *path, coreid_t
    core,
```

```
                                        domainid_t *newpid);
errval_t ump_proc_resume(struct ump_chan *chan, domainid_t pid);
errval_t ump_proc_suspend(struct ump_chan *chan, domainid_t pid);
errval_t ump_proc_mgmt_ps(struct ump_chan *chan, struct proc_status **ps, size_t *num);
errval_t ump_proc_mgmt_get_proc_list(struct ump_chan *chan, domainid_t **pids, size_t *num);
errval_t ump_proc_get_pid_by_name(struct ump_chan *chan, const char *name, domainid_t *pid);
errval_t ump_get_status(struct ump_chan *chan, domainid_t pid, struct proc_status *status);
errval_t ump_get_name(struct ump_chan *chan, domainid_t pid, char *name, size_t len);
errval_t ump_proc_mgmt_terminated(struct ump_chan *chan, domainid_t pid, int status);
errval_t ump_proc_mgmt_wait(struct ump_chan *chan, domainid_t pid, int *status);
errval_t ump_proc_mgmt_register_wait(struct ump_chan *chan, domainid_t pid, void *resp_chan,
                                     struct waitset *ws);
errval_t ump_proc_mgmt_kill(struct ump_chan *chan, domainid_t pid);
errval_t ump_proc_mgmt_killall(struct ump_chan *chan, const char *name);

#endif
```

Listing 1: UMP Channel API