CPSC 436A: Final Report

Aadit Rao
University of British Columbia

Donggyu Kim
University of British Columbia

Owen Wang
University of British Columbia

Naufal Wibawa
University of British Columbia

Introduction

This report documents the design and implementation of our Barrelfish-based operating system, developed by Group 3, Zander, as part of the CPSC 436A course. The group name "Zander" was chosen in reference to the 2025 Swiss Fish of the Year, designated by the Swiss Fishing Federation [1], as a nod to Barrelfish's origins at ETH Zurich. The report outlines the overall architecture, performance metrics, and highlights the key design decisions behind our memory management, paging, process management, and inter-core communication subsystems.

From the outset, our guiding design principle has been performance. Whenever there was a choice of data structure or algorithm, we prioritized asymptotically efficient designs, favouring balanced trees such as AVL trees to achieve $O(\log n)$ operations where simpler approaches would incur $O(n)$ costs. This led us to build an operating system that provides the core services expected of a modern OS, including physical memory management, virtual memory and paging, process management, and multi-core communication with a strong focus on clean, well-defined kernel abstractions.

System Overview

At a high level, the system consists of four layers: physical memory management, user-level virtual memory and heap management, process management, and inter-core communication. Each layer exposes an interface to the layer above while relying on capabilities and message-passing rather than shared global state.

At the lowest level, our `mm` service manages all physical RAM capabilities handed to us by the kernel. Free memory is tracked in a dual-key AVL tree indexed both by base address and by block size, which lets us implement best-fit allocation and coalescing in $O(\log N)$ time. Allocated extents are recorded separately in a "cap tree" so that `mm_free` can validate inputs and support ranged and partial frees without scanning the free list. This physical allocator is the only component that manipulates RAM capabilities directly; everyone else interacts with it through a simple "allocate/free" API.

On top of `mm` the user-level paging subsystem provides per-process virtual address spaces. Each dispatcher maintains an augmented AVL "vregion tree" of reserved virtual regions and uses it to implement `paging_alloc_*` and `paging_map_*` in $O(\log N)$ time, either picking any suitable gap or honouring a fixed virtual address. Page tables are updated lazily on demand by a user-level page-fault handler that allocates frames from `mm`, inserts mappings, and updates the vregion metadata. A `morecore` layer exposes this functionality to the C library as a grow-only heap, so `malloc` and `free` can do sub-allocations out of contiguous virtual regions while relying on paging to back them with physical memory.

Processes are represented at user level by an `init`-side process manager and a per-core dispatcher library. The process manager tracks PIDs, executable names, and parent/child relationships, and provides operations such as spawn, kill, wait, and enumeration. Spawning a new process consists of asking `mm` for frames, loading the ELF image into those frames, creating a fresh virtual address space with our paging API, and then installing a new dispatcher with the appropriate capabilities. Each dispatcher runs an event loop on the default waitset that multiplexes timer events, IPC messages, and page faults.

Multi-core support is provided by a small `coreboot` library. Core 0 allocates and reserves a 1 GB RAM region for the second core, populates a URPC payload describing all boot modules plus this RAM region, and then calls the monitor to start core 1 with pointers to the boot driver, CPU driver, `init` ELF, and URPC frame. On startup, the app-core `init` reconstructs a local `bootinfo` from this payload, forges RAM and module capabilities into its own CNodes, and initializes its paging and `mm` state as if it had booted directly from the kernel. From that point on, both cores run the same code, differ only in role (BSP vs app core), and communicate exclusively via the LMP and UMP mechanisms described above.

Communication between components is entirely message-based. On core 0, we use LMP channels between `init` and the memory, serial, and process servers that are used by child pro-

cesses. These channels are wrapped in small client libraries so that most code sees a synchronous RPC interface rather than raw messages. For cross-core communication we reuse the URPC frame established at boot and layer a UMP protocol on top. A page-sized shared frame is split into two cache-line–aligned ring buffers, one in each direction, and a per-core worker thread polls the rings, decodes headers, and dispatches requests. Higher-level services (e.g., remote spawn, PID lookup, and kill) are implemented as UMP RPCs that mirror the local process-manager operations but are automatically routed to the correct core.

Milestone 1

Data Structures

In designing our memory manager, we had to decide whether to track free and allocated memory in a single unified data structure or in separate ones. A unified design with an implicit free list is conceptually simpler, but many operations degrade to $O(N)$ time. With performance in mind, we decided to separate free and allocated memory bookkeeping with two optimized data structures for our implementation. Keeping track of free physical memory addresses could be done with either a linked list or a self-balanced tree. To avoid the $O(N)$ search/insert/remove costs of a linked list, we designed a dual-keyed AVL structure, where every free interval is represented by a single extent object that is simultaneously indexed by two self-balancing trees: (1) ordered by the start address of the free region (See Figure 1), and (2) ordered by block size (See Figure 2). In the event of a tie, the block with a smaller start address will be considered smaller.

Allocation queries will use the size-ordered tree for a best-fit in $O(\log N)$, while coalescing and adjacency checks use the address-ordered tree, also in $O(\log N)$, where N is the number of free intervals.

To complement our dual-keyed AVL free tree, we have a separate "cap tree" that records what has been allocated. Concretely, it's a two-dimensional linked list: the top-level parent list holds metadata nodes for each RAM region/capability we were provided via `mm_add`, and each of those parent nodes owns a child list containing every sub-allocation, called from `mm_alloc`, carved from that region. Each child stores its base, size, and the returned frame capability, along with a back-pointer to the parent region. This structure cleanly separates "allocated" from "free" state. This allowed our `mm_free` to quickly validate capability provenance and update metadata, as well as allowing advanced features such as ranged allocation and partial frees. Our implementation allows us to use the best-fit system of allocation, minimizing external fragmentation that usually occurs from first-fit allocators, while also staying highly efficient with an $O(\log N)$ runtime compared to an $O(N)$ runtime on a simple linked-list approach. In Milestone 1, the child collections are simple linked lists, $O(k)$ within a region where k is the number of allocated regions under a parent region, which keeps the implementation straightforward. We later discuss how this design was extended by using self-balancing trees for the child collections.

Initialization

When the system boots, the first action that happens is the initialization of the physical memory allocation interface. When the system boots, the `initialize_ram_alloc()` function prepares the physical memory allocation interface. It first calls `mm_init()` to initialize the empty memory management data structures. Then, it iterates through the physical memory capabilities supplied by bootinfo, calling `mm_add()` for each. This single `mm_add` operation populates the dual-keyed free tree with all available memory and simultaneously creates the parent-region nodes in the 'cap tree' that will be used to track future allocations.

Allocation of Physical Memory

Our core allocation lies in `mm_alloc_from_range_aligned`. The simpler entry points `mm_alloc` and `mm_alloc_aligned` are thin wrappers around it that call the range-based function with `base = 0` and `limit = UINT64_MAX` (and an appropriate alignment).

On each allocation request, `mm_alloc_from_range_aligned` first rounds the requested size up to the required alignment (at least base-page size and always a power of two). It then traverses the size-ordered AVL tree to find a best-fit candidate, i.e., the smallest free block with `block_size` $\geq$ `requested_size`. Starting from that node, it walks forward through larger blocks until it finds one that also lies entirely within the caller-specified `[base, limit]` address range. The tree lookup takes $O(\log N)$ time in the number of free extents N. The subsequent scan is typically short but is $O(N)$ in the worst case, when only one of the largest blocks satisfies the alignment and range constraints.

Once a suitable block is found, its extent object is removed from both the size-keyed and address-keyed free trees, and the block is logically split in two. Any remaining portion is reinserted as a new extent into both trees, while the portion with the caller's requested size is retyped from a RAM capability into a new frame capability and returned to the caller. Finally, we record this allocation in the cap tree as a new child node under the parent region from which the block originated.

Deallocation of Physical Memory

When the user calls `mm_free`, our system will first validate the frame capability that the caller provides. If the capability is valid, the system first finds the parent region in the 'cap tree' that contains the capability's address. Within that parent,

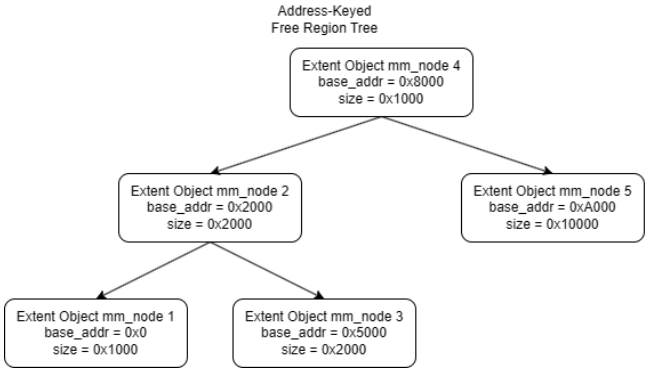

Figure 1: AVL tree ordered by block start address

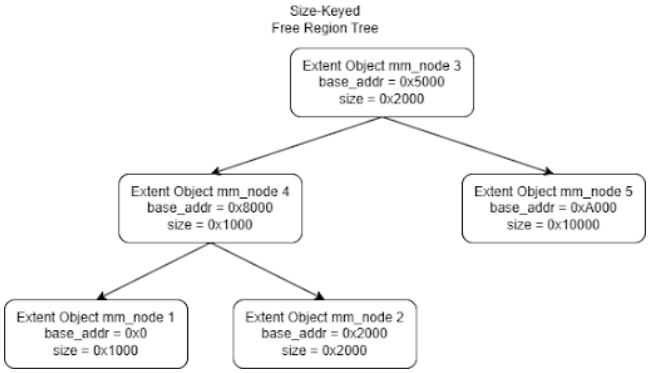

Figure 2: AVL tree ordered by block size

it then searches for the specific child allocation node. This search has three outcomes:

1. We found a child node that contains the capability given by the caller. This is a case of a full-free. The caller provides a capability that the memory allocator gives it using `mm_alloc`, and we can simply proceed to the next operation.

2. We found a child node that contains the address that the capability represents. This is a case of a partial-free, where the caller has used other parts of the capability that we give it, but it decided to give the unused portion back to the memory manager. In this case, we first free the region that the memory manager allocates for it, which we got from the child node, then immediately re-allocate the fragments that are still in use using new child nodes that point back to the same parent node. It will then use the freed region to do the next operation.

3. We did not find a child node that contains the address. This is a case where the capability that the caller gives to us does not originate from our memory manager. We simply reject this operation and return an error.

Then, using the region that we want to free, we traverse the address-keyed free region tree to determine whether there are any adjacent extents. If any adjacent extents are found, it removes them from both trees, creates a new extent object that spans all of the free region, and puts it into both trees to maintain balance. This approach separates what is allocated from what is free, keeping metadata consistent even though capabilities themselves cannot be merged. Coalescing happens in our metadata, not in the capability space.

Performance Metrics

For Milestone 1, we evaluated our physical memory allocator with two microbenchmarks that repeatedly call `mm_alloc` and `mm_free` and record the latency of each call. In both experiments we allocate a fixed-size block in a loop (8192 iterations for small allocations and 400 iterations for large allocations), then free all blocks in the same order, measuring the latency of every operation. Each configuration was run ten times. The plots report the 90th-percentile run (the 9th worst) to avoid both outliers and cherry-picking. The thin lines show raw per-iteration timings, while the bold line is a smoothed running average that reveals the overall trend.

For small allocations (See Figure 3), each iteration allocates and frees a single 4 KB page over 8192 iterations. Allocation latency is consistently low and tightly clustered: the mean is about 0.14 ms with a median around 0.11 ms, and even the

90th percentile remains around 0.15 ms. Free operations are more expensive and more variable, with a mean of roughly 1.27 ms and 90th-percentile latency around 2.3 ms, occasionally spiking up to about 7.5 ms. These periodic spikes create a saw-tooth pattern in the free curve and correspond to points where freeing a block triggers coalescing of multiple adjacent extents in our dual-key AVL free tree, which requires several tree updates but still keeps frees in the low-millisecond range.

For large allocations (see Figure 4), each iteration allocates a 4 MB block (1024 pages) over 400 iterations, for a total of 1.6 GB allocated across the run. Here, allocation latency remains extremely stable around 0.30–0.31 ms (mean $\approx$ 0.31 ms, 90th percentile $\approx$ 0.31 ms) and does not grow noticeably as the total allocated size increases, which matches the intended $O(\log N)$ behavior of our AVL tree. Free latency averages around 0.65 ms, with a 90th percentile close to 0.84 ms and a maximum just under 1 ms, again showing a regular saw-tooth pattern as coalescing events occur.

The higher worst-case latency for small frees ($\approx$7 ms vs. $\approx$1 ms for large frees) is likely due to the much larger number of extents in the 4 KB benchmark. Freeing a small block can occasionally trigger a long coalescing chain and extra metadata/slab management across many tiny intervals, a rare slow path that is far less likely to occur in the large-allocation benchmark with only 400 blocks. Overall, these results indicate that our design achieves consistently low and nearly size-independent allocation cost, while the additional work done on free to merge and maintain large, contiguous extents stays well-bounded and within acceptable millisecond-scale latency.

Further Improvement and Issues Encountered

In later milestones, we realized that `slot_alloc` frequently calls `mm_alloc` so we updated our design for representing each parent's child list. Instead of a simple linked list, each parent now owns an AVL tree keyed by address, reducing per-region lookup and removal to $O(\log k)$ where k is the number of allocations in that region, without changing any external interfaces.

Our biggest issue in Milestone 1 was that the slab allocator was never refilling correctly. We hadn't computed realistic minimum free-block thresholds for the metadata slabs (mm nodes and page-table structures), and we also lacked a re-entrancy guard, where a refill could be triggered while a previous refill was still in progress. We fixed this by deriving worst-case counts for slots/slabs along an allocation fast path, checking `slab_freecount()` against per-pool thresholds before entering operations that may allocate metadata, and guarding refills with a boolean "currently refilling" flag to prevent nested refills. The paging code now includes the pattern for refilling the slab of testing the threshold, then sets the `*_slab_refilling` flag, calls the pool's `refill_func`, clears the flag, and proceeds only if the refill succeeds. Functionally, this eliminated the bugs we observed under slab allocation pressure and made the allocator's behavior predictable.

Milestone 2

The original purpose of virtual memory is to give the ability for a process to access a larger address space than the amount of physical RAM available. Other purposes include that a program could be located anywhere in physical memory while still appearing contiguous in virtual memory. Here, virtual memory management is done in user space, rather than kernel space.

Data Structures

Our implementation of the virtual address management system contains two distinct data structures: the virtual region tree and the shadow page table. Both of these data structures live in a paging state, a virtual memory manager and paging wrapper that lives on every process. The virtual region tree represents the allocated virtual regions of that process. While a simple linked list could be used to implement the same virtual address region tracking, the $O(N)$ runtime for add/find/delete quickly becomes a performance bottleneck, as virtual address allocation and tracking are one of the most used operations done in a process. We decided to implement the tree as an augmented AVL Tree, keyed by the base address, augmented by the variable `max_free_gap_in_subtree`, which contains the maximum gap between a region's ending address and another region's starting address (See Figure 5). This is chosen so that finding a free virtual memory region, allocating virtual memory allocation, and finding an already allocated virtual memory address takes $O(\log N)$ time.

The shadow page table is required because although the user holds page table capabilities, which itself contains page table entry capabilities that can be verified by the kernel, the mapping inside the page table capabilities is opaque to the user. The user cannot see what the page table contents are, hence the need for metadata to track the entries that it adds to its own page tables, a shadow page table. Our shadow page table implementation is as follows (See Figure 6):

- `Struct pt_node` represents a page table. It holds the page table's capability, a bitmap to represent which slots are empty or occupied, as well as a head-inserted linked list to represent the page table entries, as described below.

- `Struct pt_entry` represents either a page table entry. This could be another page table in the case of an L0, L1, and L2 page tables which can hold an L1, L2, and L3 page tables respectively, or a page, which is a regular L3 page of size 4KB.

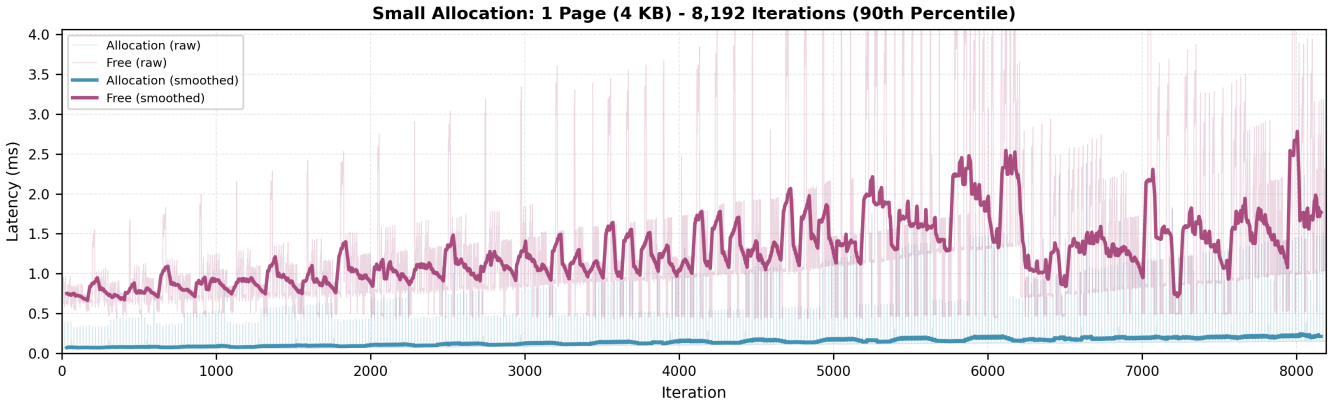

Figure 3: Small Allocations and Deallocations in mm

This results in a shadow page table traversal that is quite similar to the MMU's page table walk. Our bitmap allows a lookup into the page table to see whether a slot is occupied or not in $O(1)$, while our linked list approach works here because the number of elements is limited to 512, an $O(k)$ implementation where k $\leq$ 512 is performant enough that using a balanced tree increases complexity that is unjustifiable by the performance increase that it offers.

Initialization

After the physical memory manager has been initialized and the physical memory region has been added to the physical memory manager, the init process will start its paging system. It calls `paging_init()`, which will initialize the paging state for the init process as well as its exception stack and pagefault handler. The system initializes the init process's paging state and assigns its L0 page table as the root page table (`cap_vroot`). It then walks over the L0 page table to mark the region that is covered by existing L1 page tables, which are page tables that are used by the kernel, as guard regions in the vregion tree. Then, the system allocates 64 KB of virtual memory for the page fault handler at the address that is contained in L0 index 150, L1 index 0, L2 index 0, and L3 index 1. We chose this address because it is unlikely for a program to access this specific virtual address. Then, the system allocates 64KB of virtual memory to be used as a stack for the page fault handler. Initially we chose a fixed address to get things working, but we have since changed this so that any virtual address, given from `paging_map_frame`, that is suitable at the time of allocation is selected.

Allocating Virtual Addresses

When a program wants to reserve a virtual address space, our system contains two APIs for the program to call:

1. `paging_alloc_st` is called when the value of the allocated virtual address does not matter to the caller process
2. `paging_alloc_fixed_st` is called when a process specifically requests a virtual address region starting from a specified address.

The two functions only differ in the way they find the virtual address region to be allocated:

- `paging_alloc_st` first consults the paging vregion tree using the augmented gap variable to see the first big enough gap. If it finds one, we then proceed with the next steps. This operation takes $O(\log N)$ time.
- `paging_alloc_fixed_st` first consults the paging vregion tree using the base address key and tries to find a region that overlaps with the region that is given by the caller. If no region overlaps, we proceed with the next steps. This operation takes $O(\log N)$ time.

They will add a new node to the paging vregion tree to mark the region as allocated to avoid allocating the same region twice. Then, it sets the flags of that vregion to the corresponding flags, such as read/write/execute flags, as well as protection flags such as guard. In total, this series of operations will take $O(\log N)$ time to be executed completely. Note that these functions only reserve a range of virtual addresses to the caller, and the virtual address that these functions return is not yet backed by physical memory from a capability.

Mapping Virtual Addresses

When a process wants to access memory from a virtual address, first the virtual address itself needs to be mapped to a physical address. This is due to the fact that two processes could have some data that lives in the same virtual address, but the physical memory that is used to store the values will live in two totally unrelated places. Our operating system's user space paging supports two different types of mappings:

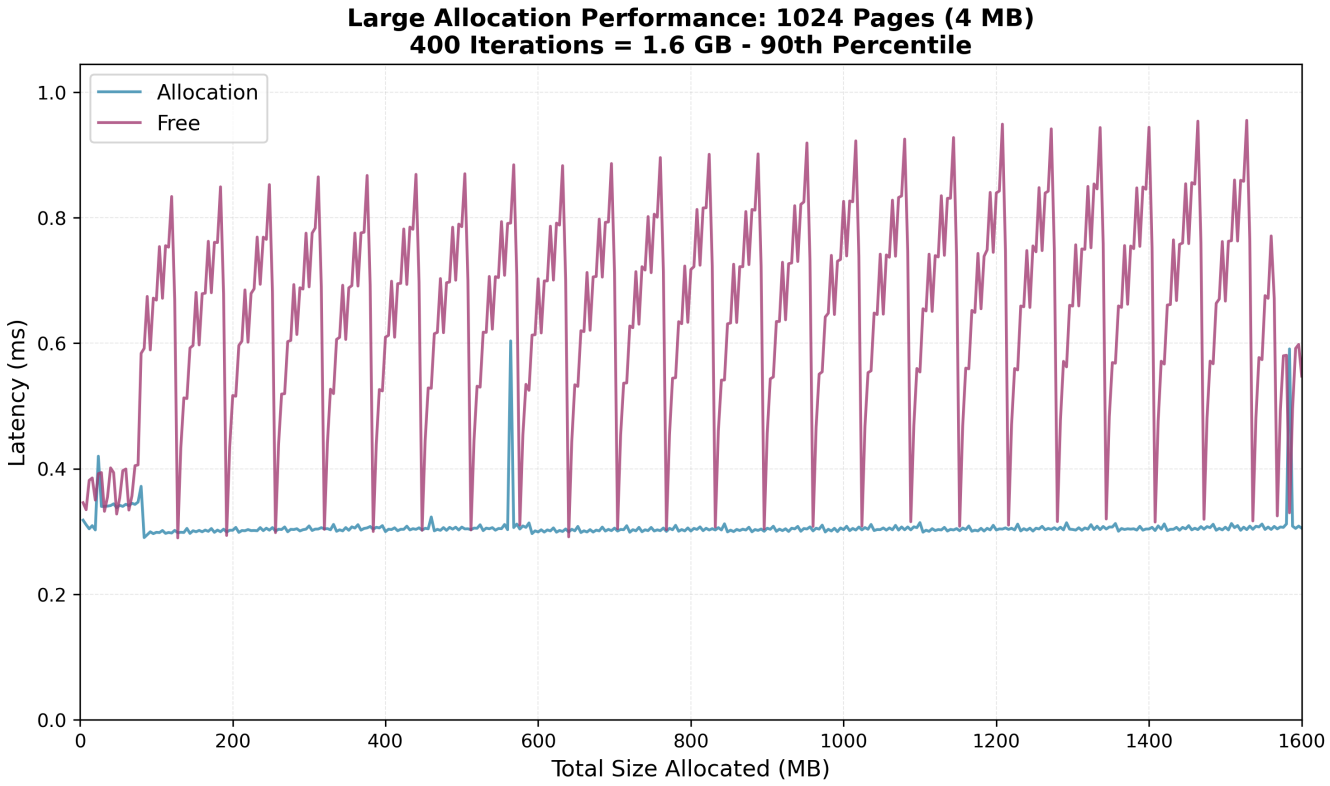

Figure 4: Large Allocations and Deallocations in mm

1. `paging_map_frame*` is similar to `paging_alloc`. It is used when the virtual address that is mapped by this function does not matter to the caller process. The flow of this function is verifying that the frame is valid, finding a virtual address region that is large enough for the size, which is done by a simple call to `paging_alloc_st`, and then mapping it to the page table by a call to `paging_map_fixed_offset_st`. This operation takes $O(\log N)$ time.

2. `paging_map_fixed*` is similar to `paging_alloc_fixed`. It is used when the caller process requires the virtual address to be mapped in a specific location. Instead of finding a free region, here we are checking whether the region has already been allocated or not. The region that is going to be mapped must not span multiple allocation regions. If there are no allocated regions, we call `paging_alloc_fixed` on the specified address. This operation takes $O(\log N)$ time.

After we have established that the virtual region is valid, we first check whether the page tables of the address exist or not. If they did not exist (we map to a virtual address that has not yet been covered by any existing page table), we create the page table. We then check whether there is an existing mapping on our shadow page table. We simply check the bitmap on the page tables at the address that we are trying to map, and if there is any overlap, we will end the operation and return an error. On M2, it first checks that if there are no overlaps, we then check the L3 page table index of the virtual address, and map the number of `MIN(number_of_pages_to_be_mapped, 511 - index)` to the L3 page table until the requested size is reached. We first traverse the shadow page table, taking $O(1)$ time. We keep track of the last-level page table, so that we do not need to re-traverse the shadow page table, and we increment the L3 page table index. If it overflows (L3 index > 511), we simply go up a level to the L2 page table and go to the next L3 page table there. We do the same in the case of an L2 and L1 page table overflow. This operation will take $O(S)$ time, where S is the number of pages that need to be mapped. As per M2, our system currently only supports mapping of base page sizes, which is the page table entry size of an L3 page table, 4096 bytes. In total, this series of operations will take $O(\log N + S)$ time, where N is the number of already-allocated regions and S is the number of pages to be mapped.

Unmapping of Virtual Addresses

After a program decides that it has finished using a region of virtual memory, it calls `paging_unmap` on the head of the

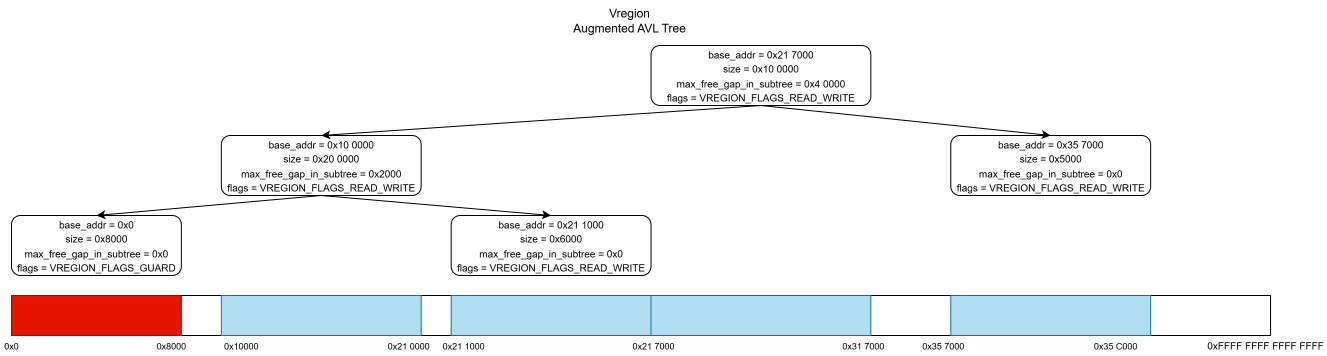

Figure 5: Virtual Region Augmented AVL Tree Data Strucuture

region that it wants to free. Our implementation requires unmapping the entire virtual region as it was originally allocated. Partially freeing a sub-range, which would require splitting an existing vregion node, is not supported. `paging_unmap` will do two operations sequentially:

1. Finds and frees the virtual address region from the vregion tree. We consult the paging vregion tree to find a region that starts at vaddr, then we check its protection bits (we don't want to remove a guard region.) After that, the system removes the region from the tree, rebalancing and updating the subtree gaps along the way. This operation takes $O(\log N)$ time.

2. Walks over the region in the shadow page table and unmaps the pages one-by-one. This operation takes $O(S)$ time, where S is the number of pages to remove.

Our implementation has it so that the shadow page table walk is only done once at the start of the unmapping, and we simply iterate to the next entry until we reach the end of the virtual address region. To detect whether an index contains a region that we will unmap, we first consult the last-level page table's bitmap for that specific slot, and if a mapping exists there, we will then unmap it. Overall, this set of operations will take a time of $O(\log N + S)$, where N is the number of allocated virtual address regions, and S is the number of pages in the virtual address region that we want to unmap.

Page Faults

When a process tries to access a virtual memory region that has not yet been mapped (through dereference, array access, or other means), the kernel will upcall to the user space's exception handler. First, we check if the address is equal to 0. This is our way to check for null pointer dereferences. Our exception handler will then check the exception type, and it will subsequently check the exception subtypes. The following paths could be taken:

- Pagefault is of permission type. The process does not have the correct permissions to access the region. We simply terminate the calling process .

- Pagefault is of translation type. This means that the mapping is not yet installed on the page table. Here, we consult the vregion tree to check whether the address that the process wants to access has already been allocated or not. If not, we return an error stating that the page fault occurs on a memory region that has not been allocated for the calling process. This operation will take $O(\log N)$, where N is the number of allocated virtual address regions.

We then check for the permission flags on the virtual address region that has been allocated. If there is no permission error, such as read/write/execute permissions or guard region conflict, we allocate a frame capability from the physical memory manager and then we map just one frame that contains the faulting virtual address. This operation will take $O(\log M)$ time, where M is the number of free blocks in the physical memory manager. If the user accesses another faulting address, it will go through these steps again. Overall, the time complexity of the page fault handler is $O(\log(N * M))$, where N is the number of allocated virtual address regions.

The Heap

Our heap implementation is built on top of the morecore interface provided by Barrelfish, which sits between `malloc/free` and our user-level paging system. Instead of using a static 16 MB heap, we implemented a dynamic heap which is backed by our paging allocator. At a high level, each `morecore_alloc` call reserved a new region of virtual address space via `paging_alloc`, while `morecore_free` tears down a previously allocated region via `paging_unmap`.

We deliberately chose to call `paging_alloc` on every `morecore_alloc` instead of reserving one giant virtual heap region and managing a sub-pool ourselves. Because the C library allocator already batches many small malloc calls into larger morecore requests, the OS only sees occasional

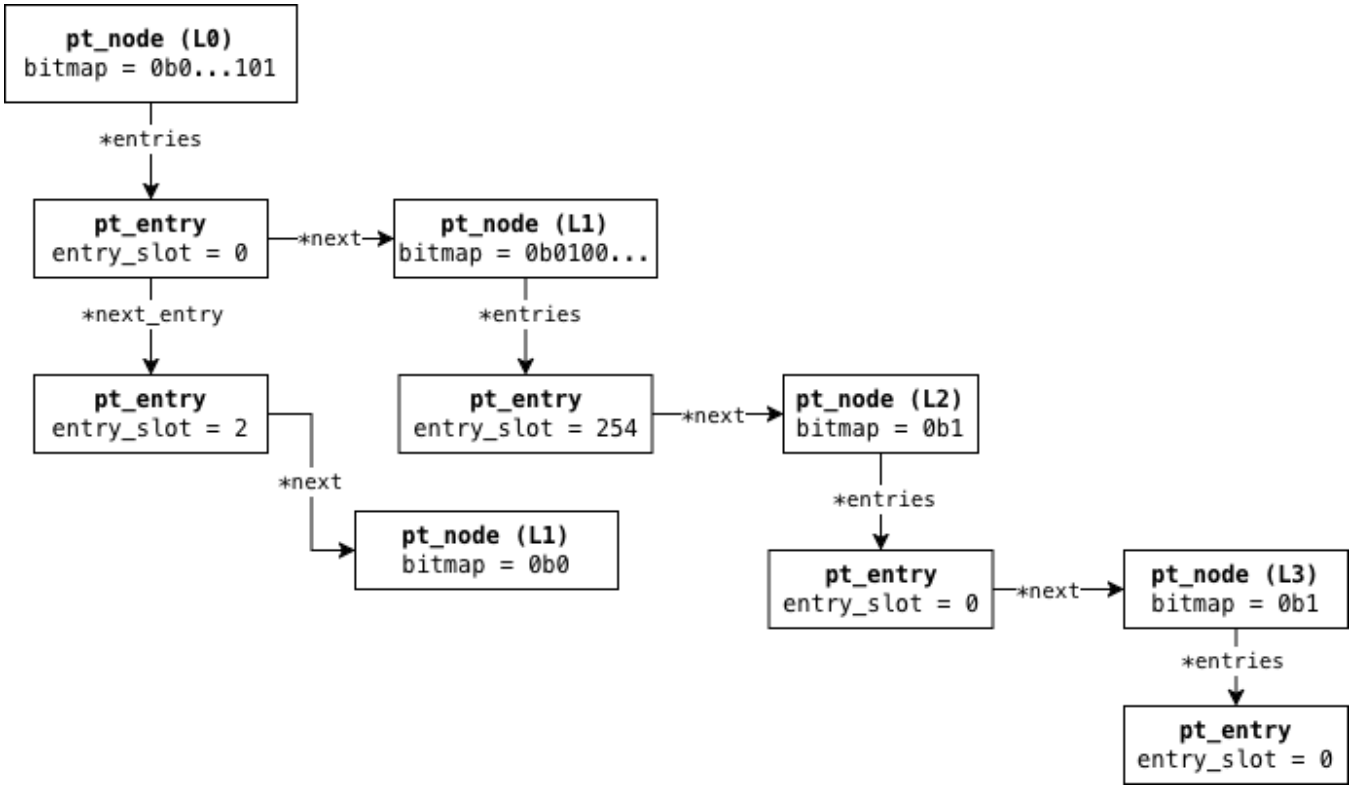

Figure 6: Shadow Page Table Data Strucuture

page-sized heap extensions instead of every tiny allocation. For example, we conducted a stress test that called malloc(4) 1024 times, `morecore_alloc`, and thus `paging_alloc`, was invoked only 8 times, with the rest of the allocations being satisfied from previously allocated chunks. This kept the heap backend simple without introducing noticeable overhead.

We also always round the requested size up to a multiple of `BASE_PAGE_SIZE`. This made sure that every heap chunk we hand to the paging system is page-aligned and page-sized, which fits naturally with how our vregion tree and shadow page tables operate. Page-aligned regions are easy to lazily back by frames on page faults and easy to unmap cleanly via `paging_unmap` when the region is returned back to us. The tradeoff is that very small morecore requests conceptually "waste" some bytes inside the last page, but those extra bytes are reused by the allocator for subsequent malloc calls, so the internal fragmentation is handled at the libc level.

Performance Metrics

For Milestone 2, we benchmarked the latency of our user-level paging on a single core. We ran microbenchmarks that repeatedly allocate or map a single 4 KB page for 8,000 iterations and recorded the per-iteration latency of `paging_alloc_st` / `paging_alloc_fixed_st`, `paging_map_frame` / `paging_map_fixed_frame`, and `paging_unmap`. Each configuration was run multiple times; the plots report the 90th percentile run to avoid both outliers and cherry-picking, with light lines for raw per-iteration measurements and bold lines for a smoothed running average.

For system-virtual allocations (See Figure 7), both "Alloc/Map" and "Unmap" stay comfortably in the sub-millisecond range across all 8,000 iterations. The smoothed curves show a roughly linear increase from a few tens of microseconds up to a few tenths of a millisecond as more regions are created, which matches the expected $O(\log N)$ behaviour of our augmented AVL-backed vregion data structure. The light saw-tooth pattern in the allocation curve corresponds to occasional slow paths where we have to allocate new paging metadata (e.g. additional page tables or slabs). Unmaps are consistently cheaper than alloc and map, since they only tear down existing mappings and update bookkeeping without touching the underlying `mm` allocator.

The mapping microbenchmark (See Figure 8) exercises repeated calls to `paging_map_frame` and `paging_unmap` on a single 4 KB page in the system address space. Latencies here are naturally higher than in the allocation measurement, as each operation must walk the page-table hierarchy and modify existing entries rather than simply reserving a fresh region. Nonetheless, the smoothed curves again grow slowly and remain well below one millisecond, indicating that page-table lookups and updates scale smoothly with the number

of mapped pages. As in the allocation benchmark, unmaps are slightly faster and less variable than maps, reflecting the fact that they benefit from already-resolved virtual addresses and avoid additional metadata allocation. Overall, these results suggest that our paging subsystem delivers predictable, low-latency behaviour for common 4 KB operations, with overheads dominated by expected tree lookups and occasional page-table growth rather than pathological slow paths.

In addition to the 4 KB measurements, we also measured the cost of allocating and mapping a 4 MB region (1024 base pages) in the system address space (See Figure 9 and Figure 10). Allocation and unmap latencies remain quite stable as the total allocated size grows, but the absolute costs are noticeably higher than in the single-page case because each operation must touch all 1024 page-table entries. This is expected from our current design, which only supports base pages and therefore performs work proportional to the number of pages in the region. With support for superpages, the same 4 MB range could be covered by a handful of large mappings instead of thousands of 4 KB entries, which would substantially reduce latency for these large allocations and mappings and lower TLB pressure in real workloads.

Further Improvement and Issues Encountered

For Milestone 2, our primary goal was correctness and clear structure rather than micro-optimizing every path. The design we implemented has worked reliably, but it also exposed several opportunities for future improvements.

First, our shadow page table still stores per–page-table metadata as a linked list of up to 512 entries. This keeps insertion simple (head insertion in $O(1)$), but lookups remain effectively $O(k)$ in the number of entries $k \leq 512$. A natural extension would be to replace this with a compact array (indexed directly by slot) so that lookup, insertion, and deletion are all $O(1)$ while keeping memory overhead low.

Second, although our M2 implementation already mirrors the MMU's walk at the L3 level, each contiguous run of pages is still represented as individual page-table entries in the shadow structure. In Milestone 3, we refine this further by using multi–entry `vnode_map` calls, making large mappings cheaper.

Finally, our current paging layer only supports base pages. Adding support for superpages would reduce TLB pressure and cut down the number of page-table entries needed for large, contiguous regions. Our existing vregion tree and shadow page-table structures are designed in a way that superpage support could be added on top.

Milestone 3

Data Structure

For process management, we maintain a single global "process table" implemented as a growable vector indexed by PID. Each entry corresponds to exactly one PID and is stored at index pid-1(to match our 0-based array), giving us O(1) lookups from PID to metadata (See Figure 11). For each process, the table stores the PID, its absolute path of the binary (which we obtained from `multiboot_module_name` function), current state, and pointer to the associated `struct proc_status` and `struct spawninfo`. When a process is killed, we never remove its entry from the table; instead, we free the allocated proc_status and spawninfo structures but keep lightweight metadata like PID, name, and final state, which allows us to answer queries about exited processes. When spawning a new process, we allocate a new PID (which is a single instance in our table), ensure the table is resized to cover that index, and fill the corresponding entry exactly once. We chose this table design because it is simple to implement and matches the fact that PIDs are unique unsigned integers. Therefore, given a PID, most helper operations become constant-time operations for common tasks such as fetching status, updated state, and cleaning up after termination. For aggregate operations such as listing all running PIDs or Statuses, we provide helper functions that linearly scan the table and filter entries based on their state.

In addition to the PID-indexed table, we also needed efficient getters for getting all PIDs running given a program name. Rather than linearly scanning the entire table on every query, we added a second data structure being an AVL tree keyed by the absolute path of the binary (again, the string obtained from `multiboot_module_name`). We considered a hash table for $O(1)$ expected lookups, but chose an AVL tree because it was simpler to implement correctly under time constraints while still giving us $O(\log N)$ operations. Each node stores the absolute path, a dynamic array of PIDs currently running that program, pointers to its left/right children nodes, and a height for balancing (See Figure 12). On spawn, we either create a new node for a previously unseen path and insert the PID, or find the existing node and append the PID to its array. The tree only tracks active or suspended processes so when a process is killed, its PID is removed from the corresponding node, and if that array becomes empty the node itself is removed and freed. For lookups, if the caller provides an absolute path we simply search by that key, but if they provide only a binary name, we traverse the tree, use C's `strrchr()` function on each stored path to get the final component, and collect matching PIDs. This design gives us logarithmic behavior for the common absolute-path case, while still supporting "by binary name" queries when needed.

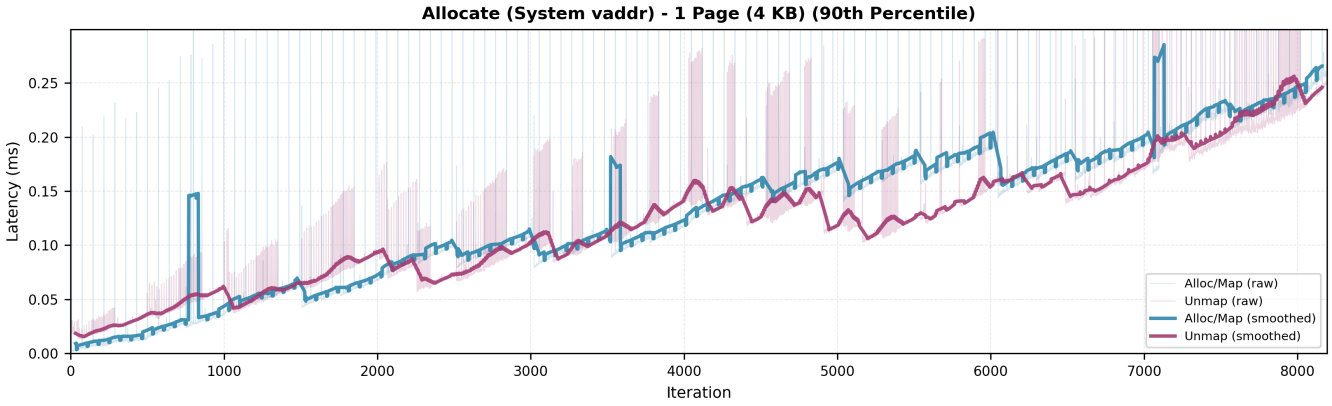

Figure 7: Virtual Memory 4 KB Allocation

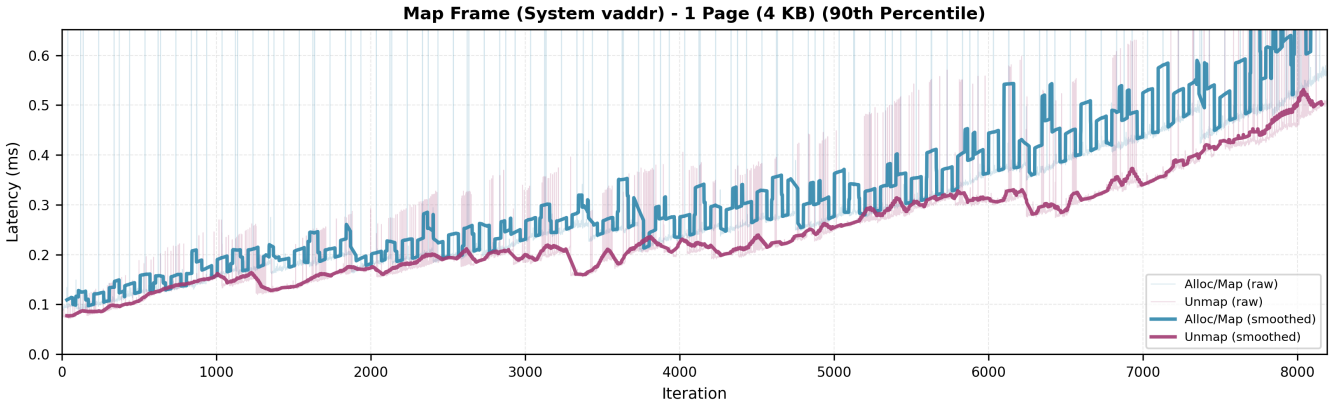

Figure 8: Virtual Memory 4 KB Mapping

Spawning Processes

The spawn library coordinates every piece of process creation, tying together capability construction, address-space setup, ELF loading, and runtime initialization so that a child can inherit just enough from its parent to start executing independently. We construct a new process around a `struct spawninfo`, which we refer to as `si`, which the parent fills via `spawn_load_with_caps`. This function prepares the child completely but does not make it runnable; the final transition is handled by `spawn_start`.

The parent is responsible for bootstrapping the child's CSpace according to the Barrelfish conventions. We first create the child's L1 root CNode and store its capability in `si→rootcn_cap`. From this root, we allocate several L2 CNodes:

- A Task CNode at `ROOTCN_SLOT_TASKCN`, populated with
  - The child's dispatcher capability (`TASKCN_SLOT_DISPATCHER`) created via dispatcher_create() into a fresh slot and then copied into the clide.
  - A self endpoint (`TASKCN_SLOT_SELFEP`) for future inter-process communication (IPC), retyped from the dispatched cap.
  - A copy of the child's root CNode (`TASKCN_SLOT_ROOTCN`).
  - The dispatcher frame (`TASKCN_SLOT_DISPFRAME`), backed by a frame allocated by the parent and mapped into the parent's VSpace so it can initialize the dispatcher structure.
  - An "early memory" frame (`TASKCN_SLOT_EARLYMEM`), a small RAM chunk used by the child during its own bootstrap.
- A Page CNode at `ROOTCN_SLOT_PAGECN`, which stores the child's L0 page table in `PAGECN_SLOT_VROOT`. The corresponding cap is recorded in `si→vroot_cap` and becomes the root of the child's VSpace.
- A User slots CNode at `ROOTCN_SLOT_USER`, which backs the child's slot allocator. We store this CNode in `si→root_slot_cnode` and use it to initialize a

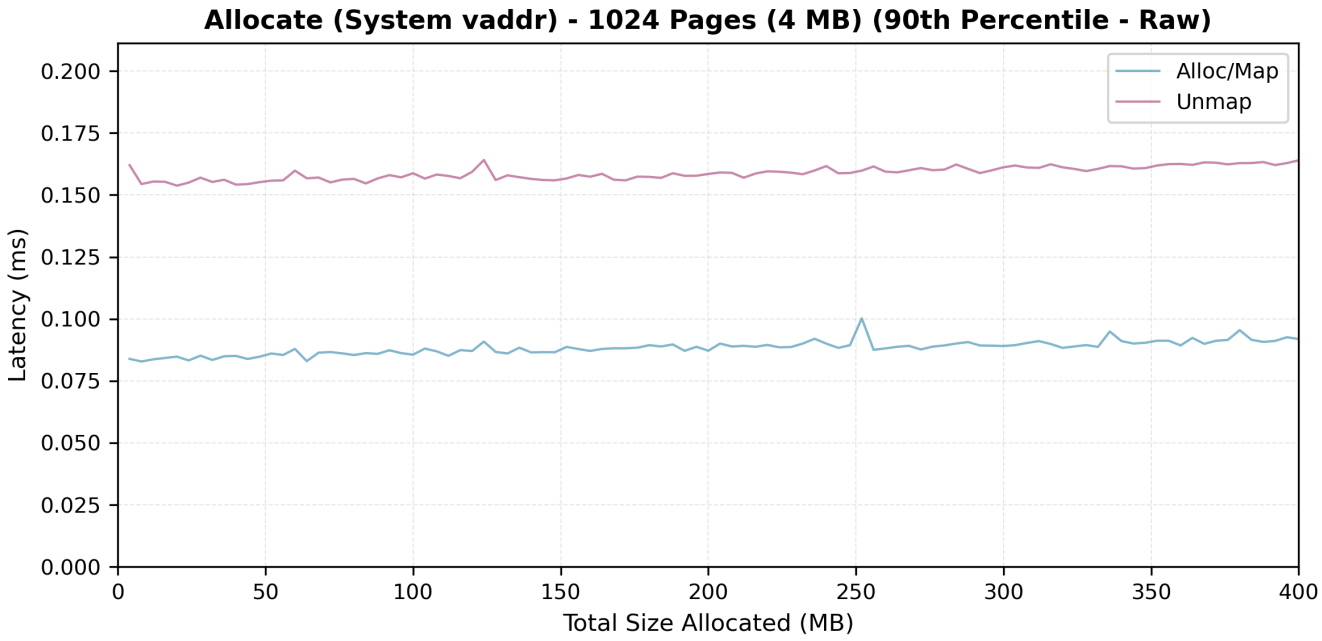

Figure 9: Virtual Memory 4 MB Allocation

`single_slot_allocator` that the child will later use to allocate its own capabilities.

We also create three additional L2 CNodes (`ROOTCN_SLOT_SLOT_ALLOC0/1/2`) reserved for future slot allocator instances.

With the CSpace in place, we initialize the child's VSpace around its L0 page table. We call `paging_init_state_foreign()` with the child's L0 vnode capability and a chosen user virtual base address of `0x0000C0000000`. In addition to catching null pointer dereferences, we settled on this value because when setting up the init process' VSpace, we observed that the first couple of L3 page tables were occupied by the kernel, causing address clashes. In retrospect, the value we chose could've been smaller. The rationale behind a better starting address is detailed in the Improvements section.

ELF loading is handled by `spawn_load_elf_image`, which wraps the generic `elf_load()` function with a custom allocator callback, `spawn_elf_alloc_cb`. For each loadable segment, the callback:

1. Rounds the segment to page boundaries.
2. Allocates a child frame capability using the child's slot allocator.
3. Allocated a parent frame capability of the same size and maps it into the parent's VSpace to obtain a local pointer.
4. Copies the parent frame capability into the child's frame slot, so both domains refer to the same physical memory.
5. Installs a mapping for that frame in the child's VSpace using `paging_map_fixed_offset_st()` at the segment's target virtual address.

The ELF loader then writes the segment contents via the parent mapping; when the copy completes, the child already owns fully populated pages at the correct virtual addresses. We also record the ELF entry point in `si→entry` and locate the global offset table (`.got`) section to compute `si→got_base` for position-independent code.

The dispatcher frame is the shared object that connects the kernel to the user process. Once the frame has been allocated and its cap stored in `TASK_SLOT_DISPFRAME`, we map it into the child's VSpace and record the child-side virtual address in `si→child_dispframe_addr`. On the parent side, `init_dispatcher_frame()` initializes the generic dispatcher structures:

- It sets the core and domain IDs (`disp_gen→core_id`, `disp_gen→domain_id`).
- It sets `disp→udisp` to the child's dispatcher-frame virtual address and marks the dispatcher as initially disabled (`disp→disabled = 1`).
- It assigns a human-readable name for debugging.

Finally, `armv8_set_registers()` configures the AArch64 register state in both the enabled and disabled save areas, installing the GOT base and entry point so that when the kernel first runs the dispatcher, the process begins executing at the ELF entry with the correct PIC register configured.

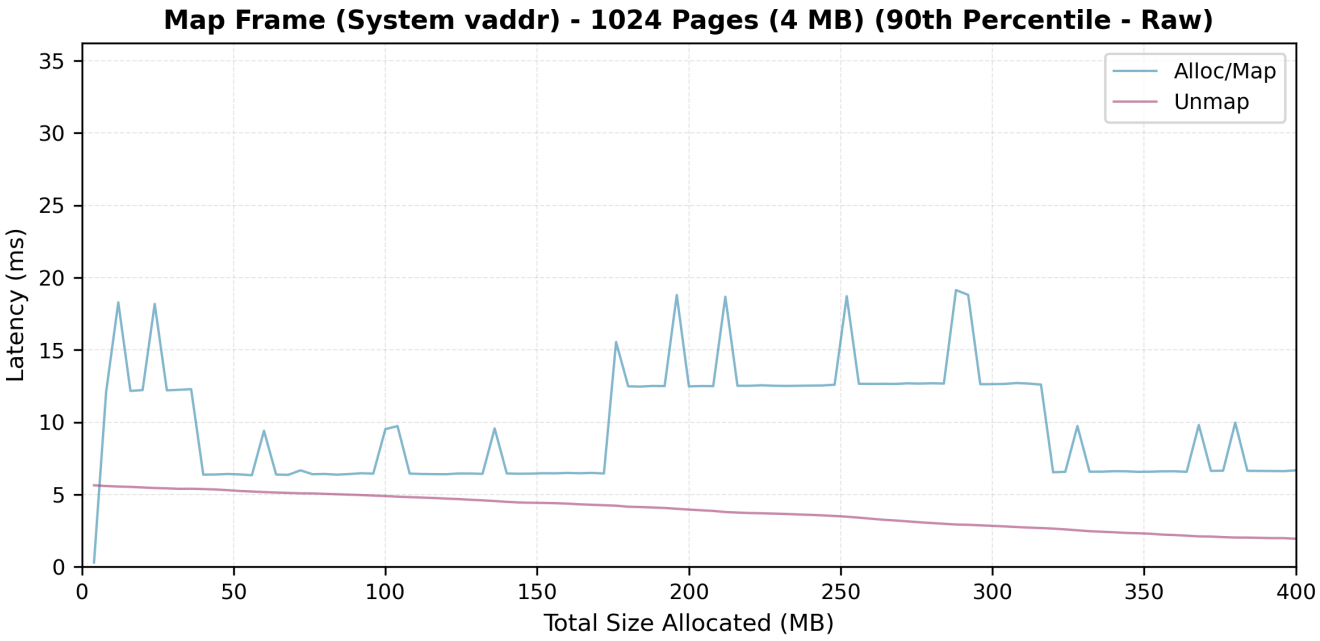

Figure 10: Virtual Memory 4 MB Mapping

index:	0	1	2	3
entry:	pid: 1 name: /sbin/init state: RUNNING proc_status*: 0x1000 spawninfo*: 0x2000	pid: 2 name: /usr/bin/shell state: SUSPENDED proc_status*: 0x1100 spawninfo*: 0x2100	pid: 3 name: /usr/bin/webserver state: KILLED proc_status*: NULL spawninfo*: NULL	pid: 4 name: NULL state: UNKNOWN proc_status*: NULL spawninfo*: NULL

Figure 11: PID table

To pass argc/argv capabilities into the child, we use a shared argument page pointed to by `TASKCN_SLOT_ARGSPAGE`. `spawn_setup_env()`:

1. Allocates an argument frame in the parent and copies the cap into the child's ARGSPAGE slot of the page CNode.
2. Maps this frame into both the parent and child VSpaces.
3. Treats the frame as a `struct spawn_domain_params` header followed by a character buffer.
4. Fills in argc and copies each argument string into the buffer, setting each argv[i] to the child-side virtual address of that string.
5. Null-terminates argv and leaves envp empty for now.

It then updates the child's enabled-register save area so that the argument-page pointer is passed as a parameter to `spawn_start()` inside the child when it begins execution.

`spawn_load_with_caps()` leaves the child in the `SPAWN_STATE_READY` state. Its CSpace and VSpace are populated, the ELF image is loaded, the dispatcher fram is initialized, and the argument page is prepared. The final step is to tell the kernel to start running it. This is handled by `spawn_start()`. The function first checks that the spawninfo pointer is valid and that the state is `SPAWN_STATE_READY`. It then issues a dispatcher invocation with the child's CSpace root, VSpace root, dispatcher frame, and the parent-side dispatcher capability. This syscall hands the kernel the capabilities it needs to bind the dispatcher to the child's CSpace and VSpace and to mark it runnable. On success, we update `si→state` to `SPAWN_STATE_RUNNING`. Later operations such as `spawn_suspend()`, `spawn_resume()`, and `spawn_kill()` reuse the same dispatcher capability to control the child's execution, but `spawn_start()` is a one-shot transition that turns a fully constructed domain into a running process.

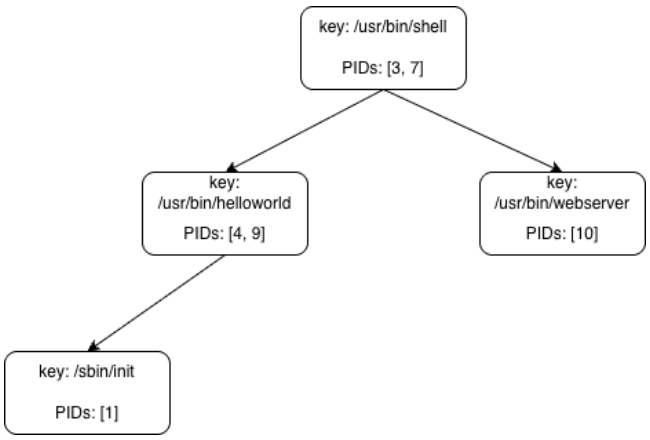

Figure 12: AVL name-pid tree ordered by name

Suspending, Resuming, and Killing Processes

At the process management layer, all control operations are implemented using our existing data structures and calls to the spawn library. For `proc_mgmt_killall`, we use our name-to-PID avl tree, depending on whether the caller passes an absolute path or a binary name, we lookup the PIDs associated with it. The function returns a dynamic array of PIDs, and we then iterate over that array, calling `proc_mgmt_kill` on each PID. We remember the first error we encounter but continue attempting to kill the remaining processes. We only return `SYS_ERR_OK` if all PIDs were successfully terminated.

For `proc_mgmt_kill`, it first validates the PID and if it's in its correct state to kill. It then fetches the associated spawninfo and hands control to the spawn library's `spawn_kill()`. If the kill succeeds, we clean up the process management metadata, update the final state, and we remove the pid from the name-to-PID avl tree. `Proc_mgmt_suspend` and `proc_mgmt_resume` follow the same pattern of validating the PID, checking the current state to make sense for the requested operation, call into `spawn_suspend` or `spawn_resume`, and on success update the PID table's state.

The spawn library for these operations is a wrapper around dispatcher operations. Each spawninfo struct tracks the parent-side dispatcher capability. `spawn_suspend`, `spawn_resume`, and `spawn_kill` simply check whether the spawninfo struct is valid and the state stored in that struct is valid, then invoke the corresponding dispatcher command. Both `spawn_suspend` and `spawn_kill` call `invoke_dispatcher_stop` on the dispatcher capability. On success, they update the state accordingly. For `spawn_resume`, it calls `invoke_dispatcher_resume` and moves the state back to `RUNNING` if the syscall succeeds.

At the kernel level, we added two new dispatcher commands, `DispatcherCmd_Stop` and `DispatcherCmd_Resume`, which are implemented by `handle_dispatcher_stop` and `handle_dispatcher_resume` in `syscall.c`. Both handlers first obtain the dispatch control block, which we call dcb, from the dispatcher capability. The stop handler marks the dispatcher as having no work, removes its dcb from the run queue via `schedule_remove`, and, if the stopped dispatcher is the current one, immediately calls `schedule()` to pick a next runnable dcb and `dispatch(next)` to context-switch away. The resume handler does the opposite where it marks the dispatcher as having work and calls `make_runnable(dcb)` to place it back on the run queue so the scheduler can pick it up again. In both cases, the syscall return registers (`x0/x1`) are filled with `SYS_ERR_OK` and a zero value, so the user-level `cap_invoke` sees a successful result. Conceptually, suspending a process simply means removing its dispatcher from the run queue, while resuming makes it runnable again. Killing is implemented as a stop plus marking the process as permanently terminated and cleaning up its user-space metadata.

Performance Metrics

Figure 13 shows the latency incurred when spawning new processes on the same core. The experiment was conducted by spawning one process per iteration, up to 25 processes in total. No spawned processes were killed/suspended. The figure shown is the result of running this test 10 times and taking the 90th percentile. The results show that the time taken to spawn an individual process scales linearly with the number of processes previously spawned, which is due to the fact that the internal data structures used for bookkeeping purposes provide $O(\log n)$ time complexities for their insertion operations, as we maintain an AVL tree to allow efficient PID lookup based on process name. We believe that if the sample size increased enough, our system should demonstrate latency increases scaling logarithmically.

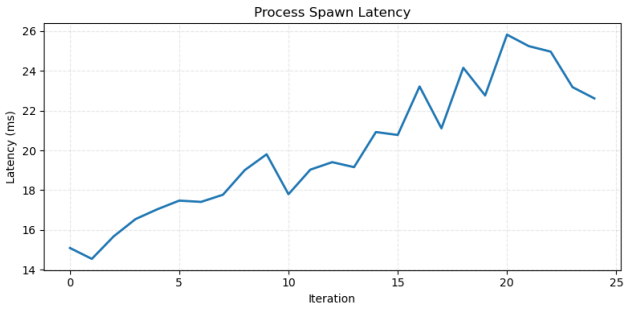

Figure 13: Process spawning latency

Further Improvement and Issues Encountered

One of the first issues we encountered in milestone 3 was the fact that we were creating too many mapping capabilities while setting up the child VSpace. This was because our paging module was initially designed to map one `BASE_PAGE_SIZE` portion of the supplied frame with each iteration. We fixed this by calculating the number of remaining page table entries (PTEs) that must be mapped, and the number of PTEs that can be mapped at the supplied virtual address (by extracting the L3 page table index from said address). This is capped at 512 to account for the maximum size of an empty L3 page table. This value was specified as an argument into `vnode_map()` (which was previously just set to 1 for frame mappings). This fix allowed for each mapping capability to refer to multiple PTEs, cutting down on the number of required CNode slots.

As mentioned earlier, the starting virtual address selected for the child process setup phase could be improved. The current address, `0x0000C0000000`, is unnecessarily high up in the address space, which means we are neglecting the regions `0x1000` to `0x0000BFFFFFFF`. One `BASE_PAGE_SIZE` frame starting at address `0x0` would be sufficient to catch null pointer dereferences. So, a better implementation would be to have addresses `0x0` to `0xFFF` as a guard region (since the smallest unit of physical memory mapping is `BASE_PAGE_SIZE`), and the base virtual address could start from `0x100000` and onwards. This approach would provide the parent process a larger child address space to be used during its VSpace setup.

In order to support foreign address mappings, our paging module logic had to be edited. By design, the capability system will not allow the parent process to invoke capability operations on caps that do not exist in its own CSpace. Our old implementation of installing page mappings only considered mapping in a process' own address space. So, we modified our paging state struct to include a boolean flag `is_foreign`, which indicated whether the paging state object represented a foreign address space. During VNode allocations, if `state→is_foreign` is true, we first call `vnode_create()` to create VNodes in the parent's address space. We then call `cap_copy()` to copy this capability into the child's address space. This logic is also used to create and transfer L3 → Frame mappings into the child's address space.

Milestone 4

Initialization

To initialize LMP for our RPC subsystem, we first set up endpoint capabilities for both the init process and each newly spawned child process. On the init side, any spawn function in our process management calls `spawn_setup_ipc()`, which calls `lmp_chan_accept()` to create an LMP channel and allocate a local endpoint for init. At this point the channel has no destination endpoint. Then we simply copy init's endpoint capability into the child's CSpace in `TASKCN_SLOT_INITEP` so that the child can later bind back to init.

When the child starts running, its early startup code in `init.c` calls `aos_rpc_init()`. This function creates the child's own endpoint and uses the INITEP capability as the remote endpoint. The child then sends an initial LMP handshake message (of type `AOS_RPC_MSG_INIT_CHANNEL`) containing its endpoint to init. On the init side, our generic RPC handler detects this special message, copies the child's endpoint into `channel.remote_cap`, and replies with an acknowledgment. After this handshake, both sides have their `local_cap` and `remote_cap` set, and the LMP channel can be used for all subsequent RPCs.

During spawn we also set up a shared "RPC argos" frame that both init and the child map into their address spaces. Init maps this frame and stores its base address in a field in our `aos_rpc` struct called `common_rpc_buf_init`, while the child maps the same frame and stores it in a field called `common_rpc_buf_child`. Our RPC divides this frame into per-request regions (which can be indexed by sequence number) to carry larger arguments and return values that do not fit into LMP payload words.

Finally, we integrate the channel with Barrelfish's event system by registering our receive handler on the default waitset (which can be obtained from `get_default_waitset()`).

On the server side, specifically in `spawn_setup_ipc()`, we register our init's LMP handler function, and on the client side any RPC call repeatedly calls `event_dispatch()` on the same waitset.

Our Structure

For this milestone, we kept our RPC structure deliberately simple and centralized. The core abstraction is a single `struct aos_rpc` that wraps the LMP channel plus any client-side state (e.g., message type / sequence numbers / pending reply info). Instead of defining separate RPC trucks and channels for the init, serial, memory, and process servers, all four services share the same `aos_rpc` instance. The different RPC "channels" (`aos_rpc_get_init_channel()`, `aos_rpc_get_serial_channel()`, etc.) simply return pointers to this common struct, and we multiplex requests by tagging each message with a type field, which is an enum for init, serial, memory, and process operations. This way, it was much simpler and more robust than maintaining four separate structs. We only had to implement sequencing, pending-request tracking, and thread-safety only once, that way we could avoid cross-channel ordering and deadlock bugs and juggling multiple bindings (See Figure 14).

We made a similar choice for event handling. All LMP traffic is gone through by a single waitset (the default waitset) and a single event loop calling `event_dispatch()`. There was the option of each LMP channel having its own waitset, but we chose one waitset per domain for simplicity and to avoid subtle ordering bugs. A single waitset means there is exactly one place where we block and dispatch events, which makes the control flow and debugging much easier.

Wait() and Exit() using LMP

For `wait()`, the client side only ever calls `proc_mgmt_wait(pid, status)`, which internally issues an `AOS_RPC_PROCESS_WAIT` RPC over our generic LMP-based `aos_rpc` channel to init. On the init side, the RPC server handles this message by calling `proc_mgmt_register_wait()`. If the target process has already exited, `register_wait` returns a special error, in which case we immediately look up the stored proc_status and send back a single LMP reply containing the exit code. Otherwise, we allocate a `waitlist_entry_t` in our PID table, where we store the caller's LMP channel pointer, its `aos_rpc` struct, and sequence number. This `waitlist_entry_t` is stored in a linked list in our PID table along with other processes waiting on this exact same process. In this case, we don't immediately send back an LMP message, since we have yet to wait for the process to finish.

When a child exits, `libc_exit()` in the child calls `proc_mgmt_exit(status)` on the client side, which turns into an `AOS_RPC_PROCESS_EXIT` to init. The process server handles this as a `proc_mgmt_terminated(pid, status)`, which updates the PID table to `PROC_STATE_EXITED` and then calls `proc_mgmt_wake(pid, status)`. `proc_mgmt_wake` walks the linked-list waitlist for that PID and, for each waiting entry, sends an LMP reply `AOS_RPC_PROCESS_WAIT` with the saved sequence number and final status, then frees the node. On the client side these replies wake the blocked calls, returning the correct exit status to each waiter.

Thread Safety and Concurrency

In our system, multiple threads can safely issue RPCs over a single LMP channel because we build a TCP-like mechanism on top of LMP. Every RPC is tagged with an 8-bit sequence number, allocated under a sequence-mutex. The sequence number is incremented and then mapped to a slot in the pending array via `seq % AOS_RPC_MAX_PENDING`. Access to this pending table is protected by pending mutex. If a slot is in use, we skip that sequence number and try another; otherwise we mark it in use, store the sequence number, clear the completion/error fields, and record any capability or response metadata. Each pending entry has its own mutex and condition variable, so the calling thread can block on "its" RPC without interfering with others.

When a thread makes a call via `aos_rpc_call` (See Listing 1), it first allocates a pending slot, constructs a message of the form `[msg_type, seq_num, error_code=0, data...]`, and sends it using the appropriate `lmp_chan_sendX`. After sending, it calls `aos_rpc_wait_for_response`, which repeatedly dispatches events until its own request is marked complete. On the receive side, our child handler function demultiplexes replies replies by sequence number, checks under our queue mutex that the slot is still in use and matches the expected `seq_num`, then under the per-request mutex fills in our response data such as error, capability, anything from our frame buffer, and marks `completed = true`, and signals the condition variable. The waiting thread wakes up, copies out the response, handles any capability, and frees the slot.

For large payloads, such as strings, argv buffers, PID lists, proc_status, etc., we use a single shared RPC frame that is logically segmented per sequence number. Both client and server compute the same offset `(seq_num % AOS_RPC_MAX_PENDING) * RPC_ARGS_MAX_DATA_SIZE`. Before sending, the client copies data into its slice and the server reads it from the corresponding slice and can also overwrite that region with results This segmentation means several threads can be doing variable-size RPCs concurrently without clobbering each other's data.

On the init side, we also avoid doing heavy work inside the LMP receive handler. The function `aos_rpc_server_handle` just receives the message, handles the special handshake, and then enqueues normal

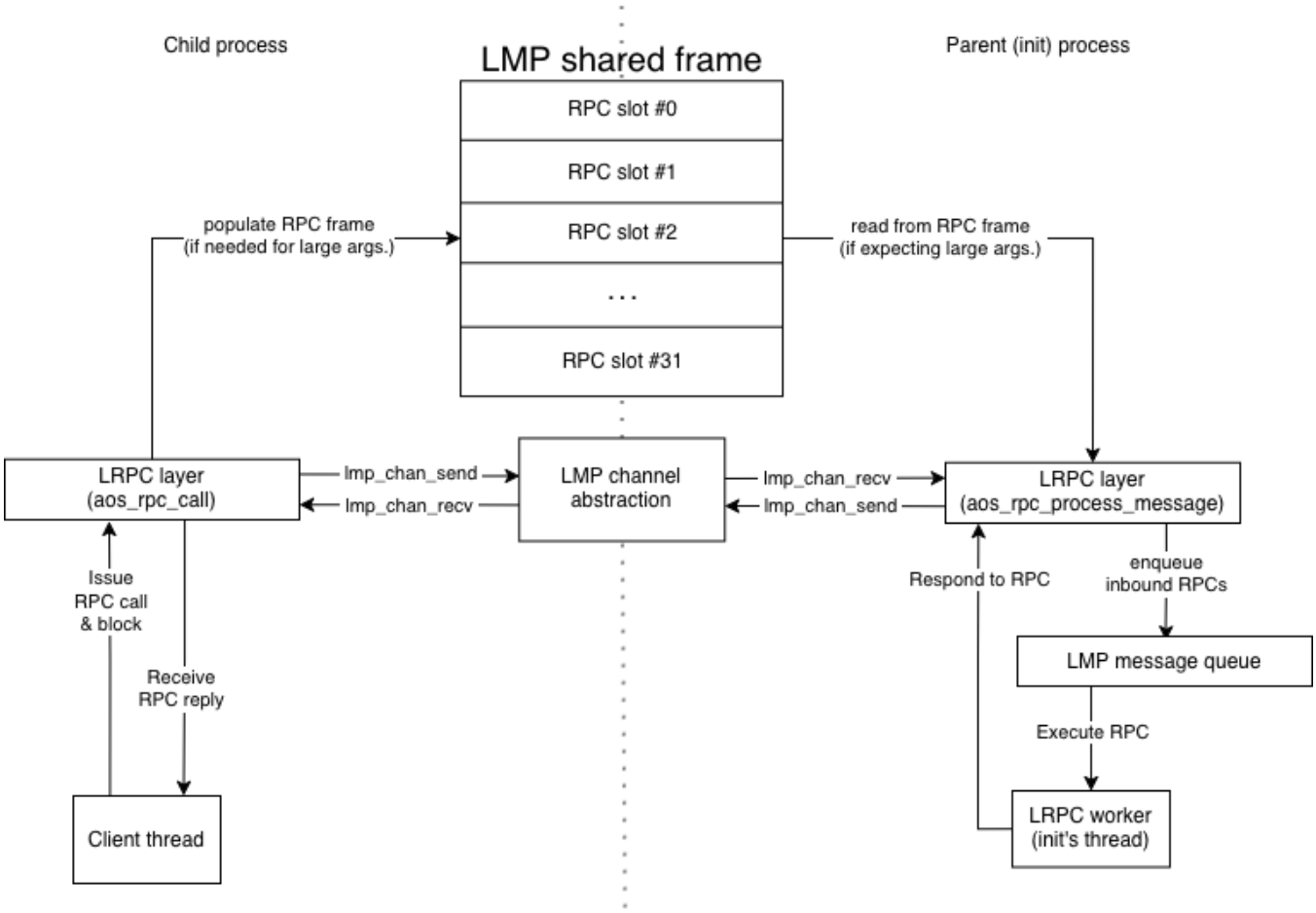

Figure 14: LMP Process

requests into a global RPC queue. This queue is protected by a mutex and condition variable, and a dedicated worker thread dequeues requests and calls `aos_rpc_process_message` to actually run RAM allocation, process management, serial I/O, and so on. Replies are sent which use the same message format and retries on transient send errors. Overall, sequence numbers and and the pending table give us safe multiplexing of replies, per-request locks and condition variables coordinate waiting threads, the segmented frame isolates large arguments, and the server-side worker and queues keep the receive path responsive even when multiple clients are issuing RPCs in parallel.

Performance Metrics

Three RPCs were chosen for measurements: `send_string`, `get_ram_cap`, and `proc_spawn`. Not all RPCs incur the same overhead (sending/receiving new capabilities, populating the RPC ringbuffer, etc.), and we felt that these 3 RPCs best represented the different setup overheads at the RPC/LMP layer.

We measured the latency of the `send_string()` RPC to see how long RPC calls took to complete (See Figure 15). To measure latency under different loads, we issued RPC calls with 16, 128, and 1024-byte string payloads. The CDF shows that the 16-byte RPC calls all had $<$0.06ms round-trip latency, while all 128-byte RPCs completed within 0.3ms. To our surprise, roughly 96% of the 1024-byte RPCs finished under 0.35ms, which wasn't too far behind their 128-byte counterparts. The likely reason is that both 128-byte and 1024-byte RPCs require the use of the shared RPC ringbuffer to transfer the entire payload, since our implementation of LMP only allowed 32 bytes of free payload on a 64-bit architecture.

In addition, we measured the round-trip latency of the `get_ram_cap()` RPC (See Figure 16). The green dotted line is the average latency for local RAM allocations, which is similar to the performance observed in our discussion of Milestone 1. The blue trendline shows the average latency of remote RAM allocations (ie. requesting RAM over RPC). RAM allocations via RPC seem to be roughly 5ms slower than its local-allocation counterpart. Another interesting observation is that latency seemed to increase a fair bit, up to

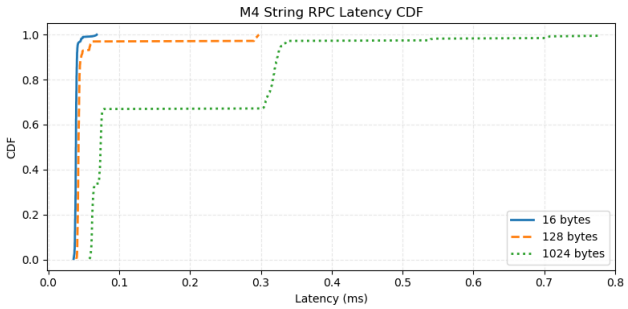

Figure 15: Latency of the `send_string()` RPC

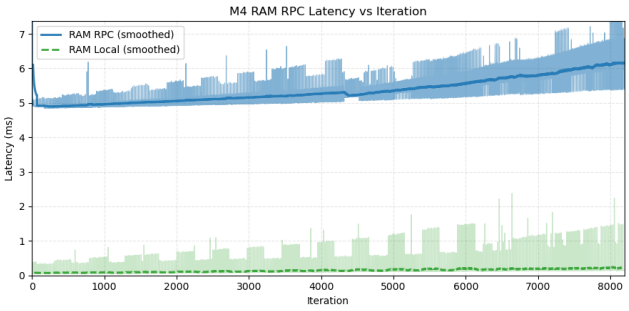

Figure 16: Latency of the `get_ram_cap()` RPC

roughly 6ms, by the time we invoked our 7000th RAM RPC, but the reason for this is unclear.

Next, we measured the latency associated with RPCs that utilized `proc_mgmt`'s spawning operations. For this specific test we utilized `aos_rpc_proc_spawn_with_cmdline`. The green dotted line is the same measurement taken in Milestone 3's performance investigation (See Figure 17). The RPC version has an added overhead of roughly 6ms, consistent throughout all 25 iterations. The same trend from Figure 16 is evident in their RPC versions: other than the flat 6ms-overhead, we see an increase in latency as more processes are spawned.

Further Improvement and Issues Encountered

In our initial LMP design we implicitly assumed that only one RPC would be in flight per channel. As soon as we started calling the LMP RPC's from multiple threads at once, replies could race. For example, a response might wake up the wrong caller, or be processed while no one was actually waiting for it. This is what led us to introduce the aos_rpc_pending "queue" table, per-request mutex, and TCP-like sequence numbers on top of LMP.

Concurrency around the waitset was another problem. We originally called event_dispatch() directly from multiple threads without any protection, which could lead to nested dispatches and potential guard corruption in our `aos_rpc` struct. This is why we added a waitset mutex and a depth counter where the outermost caller takes the lock, and recursive dispatches reuse the same waitset safely. Getting that right took a few iterations.

One improvement we would like to add if we were given more time would be that our flow control is very coarse. If our outgoing queue for LMP requests is full, i.e., we've reaching `AOS_RPC_MAX_PENDING`, we simple fail and return `SYS_ERR_LMP_TOO_MANY_CONCURRENT_REQUESTS`. A more polished design would be to implement queuing where we can either block new callers until there is space, or maintain a bounded queue per channel and apply fair scheduling so no single client can starve others.

Finally, our current implementation multiplexes all services (init, serial, memory, process) over a single generic `aos_rpc` channel. This is simple, but it means all RPCs contend for the same locks and sequence number space. A further improvement would be to keep the shared infrastructure (sequence numbers, pending table) but allow multiple logical channels or per-service channels under the hood, which could reduce contention and make performance more predictable under heavy load.

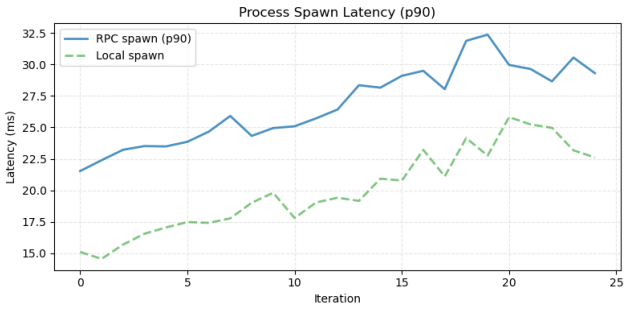

Figure 17: Latency of the `proc_spawn()` RPC

```c
/**
 * @brief Generic thread-safe RPC call with sequence numbers
 *
 * @param[in]  rpc          The RPC struct
 * @param[in]  send_words   Number of words to send (not including seq_num)
 * @param[in]  send_data    Data to send (will be prefixed with seq_num)
 * @param[in]  send_cap     Capability to send, or NULL_CAP
 * @param[in]  lmp_flags    LMP send flags (e.g., LMP_SEND_FLAGS_DEFAULT, LMP_FLAG_GIVEAWAY)
 * @param[out] recv_words   Number of words received
 * @param[out] recv_data    Buffer to receive data
 * @param[out] recv_cap     Received capability
 * @param[in]  rpc_buf_addr Pointer to a buffer containing data to copy into shared RPC frame
 * @param[in]  rpc_buf_size Size of rpc_buf_addr
 * @param[out] recv_rpc_buf Pointer to the correct offset of the shared RPC frame, based on seq_num
 * @return SYS_ERR_OK on success, error on failure
 */
static errval_t aos_rpc_call(struct aos_rpc *rpc, size_t send_words,
                             const uintptr_t *send_data, struct capref send_cap,
                             lmp_send_flags_t lmp_flags,
                             size_t *recv_words, uintptr_t *recv_data,
                             struct capref *recv_cap,
                             void *rpc_buf_addr, uint16_t rpc_buf_size,
                             void** recv_rpc_buf)
{
    // ...
}
```

Listing 1: Thread-safe RPC call helper

Milestone 5

Booting Up Second Core

To boot the second core we mostly follow the course book flow, with two small extensions. We also pass over 1 GB of RAM and all module metadata via a URPC frame. On core 0, `bsp_main` calls `coreboot_boot_core`, which (1) allocated a new KCB, (2) loads and relocates the boot driver and CPU driver ELF modules, and (3) allocates frames for the core-data struct, the new core's stack, the init binary, and a URPC frame. We then reserve a 1 GB RAM region with `ram_alloc`, record its base and size in a `mem_region`, and pack this plus all module regions from bootinfo into a `crosscore_data` structure stored in the URPC frame. Finally, we fill in the `armv8_core_data` (stack pointers, CPU entry, memory ranges, URPC frame base/length, KCB), we flush caches, and call `invoke_monitor_spawn_core` to actually start core 1.

On core 1, main detects that `my_core_id != 0` and jumps into our `app_main` function. There we map the URPC frame, read back the `crosscore_data` payload, and reconstruct a local bootinfo. For module regions we recreate the caps with `devframe_forge` and rebuild the module-names frame; for the RAM regions (including the 1 GB chunk we reserved) we call `ram_forge` and add them to the app-core memory allocator via `initialize_ram_alloc(bi)`. After that, we initialize the UMP/URPC channel using the same URPC frame, so both cores have a working allocated and a shared communication channel.

Milestone 6

Initialization

We reuse the URPC frame allocated during core boot as the shared medium (See Figure 18). In `ump_channel_init` (usr/init/main.c) we treat the 4 KB page as two 2 KB halves: core 0 writes into half 0 and reads from half 1, while core 1 does the opposite. This gives each core a fixed "local" and "remote" ringbuffer. Each ringbuffer stores TX/RX indices, sender/receiver core IDs, and 31 cacheline-aligned slots, plus a `ready_flag`. On startup, `ump_reset_buffers` clears the indices and header-valid bits in both halves to avoid consuming stale traffic. Once the app core has mapped the URPC payload, it sets `ready_flag = 0xDEADBEEF` via `send_init_complete_msg` as a lightweight "I'm up" signal after its half is clean.

After the BSP has zeroed the URPC buffers during boot and the app core has reconstructed its bootinfo from the same frame, both sides call `ump_register_channel` to attach the channel to the default waitset and start the service thread. Because the URPC frame is already mapped on both cores, no additional capabilities or copying are required to bring up the UMP transport.

UMP Process

Each core runs a single worker thread (`ump_worker` in `lib/aos/ump.c`) that continuously polls `ump_process_rx`. This function moves forward the remote TX counter, parses headers via `ump_parse_header`, and either dispatches incoming requests or completes pending replies. Every message carries an `is_req` flag, an 8-bit sequence number to represent if the message is a request or a response, and a 5-bit opcode. Replies reuse the same sequence and store an error code in word 1. Request opcodes are resolved through a handler table installed in `ump_service_start`. Our default `proc_req_handler` implements process RPCs such as spawn, status/name lookup, PID enumeration, pause/resume, exit/kill, and killall by delegating to the process manager and marshalling small results into the 7-word payload.

Replies are sent using `ump_send_response`, which enforces backpressure on the 31-entry ringbuffer by waiting until `tx_local - remote->rx_local < 31` before writing. It then fills the payload first and fills the header with appropriate memory barriers to keep both halves coherent. Outgoing RPCs use `ump_send_request_words` where the caller allocates an 8-bit sequence number (wrapping at 255), writes the request into the next local slot, and then waits (with `thread_yield`) until `complete_response` marks the single global `pending_req` as finished. At most one UMP RPC is in flight per core, and replies are matched on `(seq, opcode)`, with mismatches dropped, and the caller finally copies out the buffered payload along with the error code returned by the remote handler.

Performance Metrics

Due to lack of hardware clock synchronization between cores (CNTVCT_EL0 difference), we measured the 'End-to-End Spawn Latency' for M6 from the caller's perspective by polling the process manager status. This captures the full cost of the spawn syscall, UMP, and scheduler activation.

Figure 19 shows our test spawning processes on a remote core. Again, note that the cross-core latency (90-120ms) includes the overhead of the synchronous polling loop. This involves multiple UMP round-trips to the Process Manager to query status, as well as scheduler quantization delays. The raw hardware cost of the cross-core spawn is likely lower, but the observed latency reflects the end-to-end time for the parent process to receive confirmation of the state change.

Computed 90th percentile (p90) latencies and deltas:

p90 (ms): M6 = 117.421, M4 = 30.321, M3 = 24.915
Differences:
M6 - M4 = 87.099 ms
M4 - M3 = 5.406 ms

Further Improvement and Issues Encountered

Our UMP path is functionally correct but still fairly conservative. The largest limitation is that we only allow one in-flight RPC per core, backed by a single `pending_req` slot. This greatly simplified correctness (especially around sequence wrap-around and retrying on dropped messages) but leaves throughput on the table. A better extension would be to reuse the LMP-style pending table and support tens of concurrent UMP RPCs matched purely by `(seq, opcode)`.

References

[1] SWI swissinfo.ch. Zander named Swiss Fish of the Year. www.swissinfo.ch/eng/alpine-environment/zander-named-swiss-fish-of-the-year/88666630, January 2025.

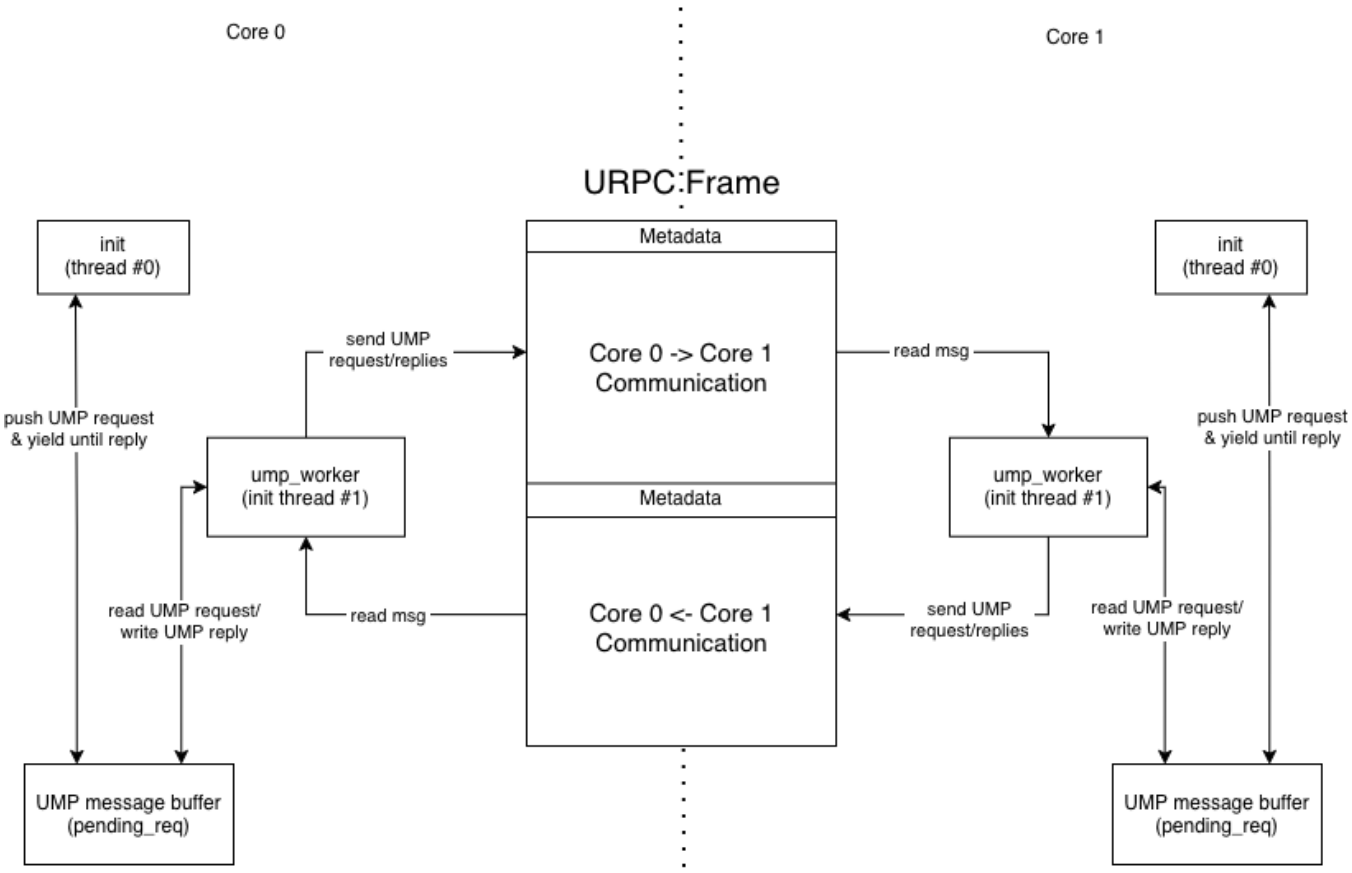

Figure 18: Flow diagram of a typical UMP call/reply cycle

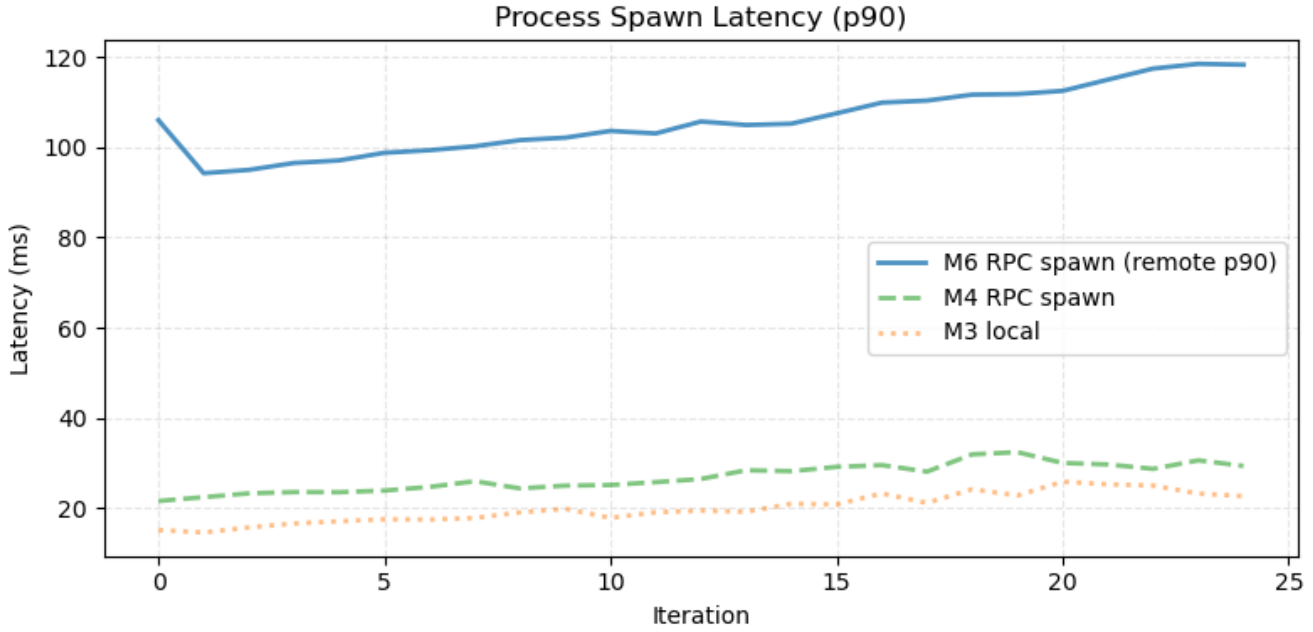

Figure 19: Latency of cross-core processing spawning