OS Capstone Final Report

Andrew Makin | Daniel Rashevsky | David Strupinski | Peter Hale

Milestone 1

Memory Management and Capabilities

Responsibilities and Results

For this milestone, Andrew drove with everyone else helping with interpreting the book and Peter and Daniel did some occasional bug fixes and implementations from home. We finished this milestone on time and implemented two of the bonuses, each worth 2 points: allocating a range of memory and partial freeing. We also added many new tests to check for any edge cases and made sure all these tests passed along with the provided tests.

Design Process

In completing this milestone, there were a few major design decisions that we came across. The chief of these was the data structure that we used. Although more ornate or performant options were available, we chose to keep a linked list of every memory block. This decision was made with primary regard to simplicity, as keeping a simple list meant we could keep the memory blocks in order, simplifying the splitting and coalescing of blocks. Although the linked list approach means that we service most calls in linear time, we did make a few optimizations. For example, we kept track of the free and total memory separately so we didn't have to traverse the entire list each time we wanted to query those values.

When implementing freeing, we ran into an issue relating to the monitor. Specifically, calling cap_revoke() would fail, as was warned about in the book. To mitigate this, we instead called cap_destroy(), which didn't seem to fail. Another issue we ran into with freeing was that we needed a capability to the original memory capability to check whether or not we could coalesce. Originally, we kept a linked list of the capabilities from the bootinfo struct, although we eventually found this to be wasteful and instead chose to store the capref to the backing memory in the linked list nodes themselves for easy access. When implementing partial frees, we split out the coalescing code into helper functions. This reduced redundancy and improved maintainability.

One other issue we ran into was that of refill cycles. We found that refilling our slot allocator would require one or more slots in itself, resulting in a cycle. To solve this, we created a separate check and refill method that we could call at various points in the code. This method would check if the slot allocator had fallen below a certain threshold of slots remaining and would refill it if so. This solved our problems with the refill cycles.

Once our code was written, we set about testing it. Because we found the provided tests lacking, we started by modifying them to do things like allocate more memory or allocate memory multiple times. When we found that this broke the autograder, we instead began to write new tests that would run after the original tests. This had the provided benefit of making sure that the original tests were still passing. We carried this approach forward to the remaining milestones.

Milestone 2

Page Fault Handling and Virtual Memory

Responsibilities and Results

For this milestone, Andrew drove for the majority, David drove at times as well, and Daniel and Peter worked alongside both of them, often leading the thinking on design suggestions and bug fixes. Significant individual work was done by Andrew and by Peter. At the time of grading, all the expected functionality was working including mapping of arbitrarily large memory chunks, and mapping at a fixed location. In addition to the base functionality, the extra functionality of unmapping of pages was implemented as well. Later on in the project we discovered a corner case error described below which had to be fixed.

Design Process

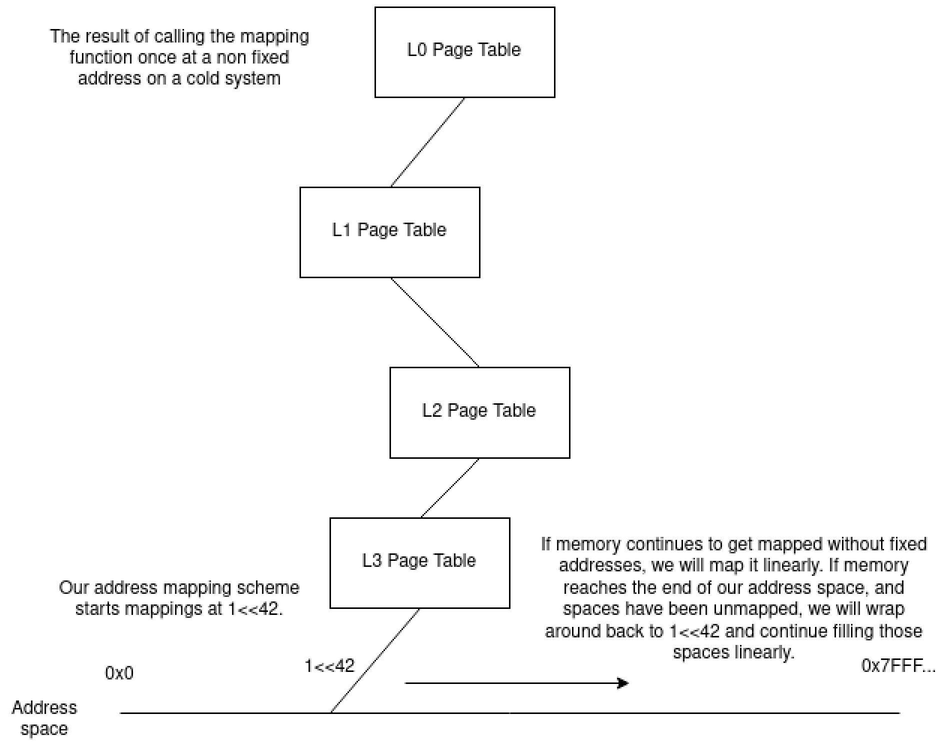
We took a while to settle on a design for this milestone. In particular, the group was split between whether we should always require a certain address to be deterministically mapped to a particular page table, or whether it would be reasonable to store the mappings of non-contiguous addresses in adjacent slots of our data structures.

We eventually settled on the first design option shown below in Figure 2.1. This means that we have one L0 page table, which is responsible for the entire address space. It has 512 slots each potentially containing a mapping for an L1 page table, so the 0th slot of the L0 page table would be responsible for mapping the addresses spanning the first 1/512th of the address space.

The first quarter of our address space is not available for mappings from the process, it contains things like mappings for the process's ELF image that are set up during spawning. The addresses are allocated linearly unless a fixed mapping is desired. Initially only the L0 page table is initialized and it contains zero mappings. Each time a new mapping is requested we use the address of the mapping and the size of the mapping to determine if new page tables will need to be created. On the first mapping, this is always the case so an L1, L2, and L3 page table are created.

The drawbacks of this design are that if we want to map two addresses that do not belong to the same L3 Page table, we will always need to map them in separate page tables—even if there would have been room in a single page table. However our design is preferable since it is far easier from a programming perspective to determine whether an address is already mapped. We also expect there to be some benefit to programs that read from their mapped memory in sequence since the mappings containing nearby addresses will always be stored together in the same page table, and therefore once one is loaded into the cache, its neighboring addresses will also have been brought into the cache.

Figure 2.1 Page table configuration after an initial mapping


The result of calling the mapping function once at a non fixed address on a cold system
L0 Page Table
L1 Page Table
L2 Page Table
L3 Page Table
Our address mapping scheme starts mappings at 1<<42.
If memory continues to get mapped without fixed addresses, we will map it linearly. If memory reaches the end of our address space, and spaces have been unmapped, we will wrap around back to 1<<42 and continue filling those spaces linearly.
0x0
1<<42
0x7FFF...
Address space

Our unmapping functionality does not deallocate page tables when possible. For example if we unmap the only mapping in an L3 page table which itself is the only mapping in an L2 page table, it should be possible to delete those L2 and L3 page tables. However, we do not do this. Our reasoning is that they are likely to be used again, and the cost of deallocating would likely be more expensive than the gain in freed resources.

Our design is required to allow mappings below 1<<42 when a process is being spawned. This is relatively easy to implement, vspace_map_foreign() has access to map only below 1<<42 and vspace_map() can access 1<<42 and above only. This restriction on vspace_map_foreign() to below 1<<42 prevents there from being anything unexpected already mapped in the new processes address space once it initializes its address space.

The virtual address space management system is required to interact with basically every other module within the operating system to the point where it is quite silly to even think of it as a module in its own right. It is better thought of from the layered design perspective, although there are not many layers to it. Whenever new memory is desired, the first layer is to request physical memory from the physical memory manager described in the first chapter. Once a capability to the memory is obtained, it can be mapped to actually give the physical memory a virtual address in the processes address space. This second layer is the functionality that our virtual address space manager is responsible for. These two layers are the only essential layers, but they can be abstracted away by a process that doesn't want to implement the interaction between the physical memory manager and the virtual address space manager. The malloc abstraction can be thought of as a sort of book ending layer; the process requests that malloc gives it more memory, and if necessary it performs the two layers described above, returning a pointer to the memory it made available.

Verifying whether a given address space is mapped or not is a constant time operation for our design: simply traverse to the correct L3 page table and check if anything is mapped in the slot reserved for that address. However, finding a free address is potentially O(N) with our design, although in many cases it will be constant since the next free address will likely be right after our last mapping, this is not always the case when the address space becomes fragmented because of fixed mappings getting in the way, or because of mappings being unmapped. In this scenario a different method of keeping track of available addresses would be preferable. Perhaps storing an array of booleans representing whether each address is free and setting array[address] to false when it is mapped would be better since we could quickly identify (maybe with a binary search) a free address to be used. However, this would create memory overhead since we would need to store all the free addresses out of a vast address space.

In the average case that we expect our OS to be used for, we do not expect it to take very long to search for an address because of the way that malloc functions. It does not map at fixed addresses, reducing fragmentation. Although calling free() can trigger an unmap, we expect these unmapped pages to be relatively evenly distributed across our address space, so it is unlikely that a program would have a virtual address space in which it has to look very far before finding an available address. Because of this, and the overhead and difficulty of implementing a list of free addresses, we went with our current design.

It would be beneficial to measure the time it takes to search for free addresses in the case when the address space is fragmented, however because of the difficulty in creating this case, and the business of finals week, we elected not to do that.

Milestone 3

Processes, Threads, and Dispatch

Responsibilities and Results

For this milestone, Daniel and Peter drove the initial design and implementation of the child process spawning and ELF loading, with Andrew and David doing some heavy debugging and finishing up the implementation. By the end of this milestone, our child process spawner could setup a child's vspace and dispatcher, load an ELF binary from a multiboot module correctly into memory, pass in command line arguments and capabilities to the child, and dispatch the process. However, we didn't implement the extra suspend/resume functionality.

Design Process

The first thing we implemented was paging_init_state_foreign in paging.c, to allow the parent process to create a page table for the child process. This involved allocating a new paging_state struct, setting up some metadata, and slab allocating space for the L0 and child nodes of the new table. This structure would support all future vspace mappings that we will perform to setup the child process. Then, it was time to start implementing spawn_load_with_bootinfo in spawn.c. This function creates the child process's spawninfo struct, vspace, and cspace/caps, before dispatching the process. The difficult part of the milestone was ensuring that all of the correct metadata and capabilities for the child process were in the right places, so this took several iterations to get right.

We started off by finding the ELF memregion for the executable we want to load, creating the L1 and L2 cnodes for the cspace, creating the child paging state struct, and mapping its third virtual page (the second holds command line arguments) to a frame in memory holding itself using paging_state_init_foreign() and paging_map_fixed_attr_offset(). The struct would be created on the parent, but inherited by the child, allowing us to only update the struct once before the process is dispatched. This approach solves the chicken and egg conundrum of the parent needing to create the vspace for the child but use it simultaneously to allocate things for that child. The first page of the child process was left unmapped, in order to allow for NULL references to segfault properly. For our process spawning implementation, we started the actual mapping 16 pages in. We aimed to map our own data in under a quarter of the vspace. That way, everything above a quarter of the vspace can be used by the process.

At this point, we started implementing process management above the level of a single process, which is handled by the init process in Barrelfish. In proc_mgmt.c, we began proc_mgmt_spawn_with_caps(), which calls spawn_load_with_caps() and eventually spawn_load_with_bootinfo(). proc_mgmt_spawn_with _caps() finds an ELF module that is passed in, creates an elf_img struct containing metadata and a capability to the ELF module in memory, finds its command line arguments, and passes this information to spawn_load_with_caps().

To keep track of our processes, we added a field to the the top of the proc_management.c file to hold the root of a linked list of struct spawninfo items representing the current processes. We modified the struct spawninfo to contain a next pointer. We added new struct spawninfo items to the beginning of the linked list so that starting a new process would be an O(1) operation. The PID for a newly-added process is one more than the PID of the last process. At this point in our implementation, we were able to attempt a test of the above functionality. We found an issue where the spawn library wasn't being properly linked in, and modified the Hake configuration accordingly.

As we iterated on spawn_load_with_bootinfo(), we continued adding more required capabilities to the cspace, ram allocing and creating cnode space for them as needed. These included several ROOTCN nodes, as well as the TASKCN nodes that fell underneath them. One of these, TASKCN_SLOT_EARLYMEM, was set to 256 pages due to another area of the code depending on it being this size. We also needed a space in the child cspace to map the ELF into. We chose slot 23 (TASKCN_SLOT_ELFMEM). Lastly, we mapped some of these objects into the parent vspace, such as TASKCN_SLOT_ARGS and TASKCN_SLOT_EARLYMEM, so we could write information to them. TASKCN_SLOT_ARGS would be accessed by a loop, which took command line arguments and copied them into the frame. TASKCN_SLOT_EARLYMEM was used by the ELF library as a destination to copy ELF code sections into the child process.

The ELF section allocator was a relatively straightforward function to write. It needed to take an ELF section's offset in virtual memory and size. Then, it ram alloced space of the correct size, and mapped it into both the parent's and child's virtual memory. This would allow the parent to fill it with code and the child to execute it. Once the ELF was relocated into the child's vspace, the dispatcher and dispatcher frame could be copied into the child's space. This involved copying the TASKCN_SLOT_DISPATCHER and TASKCN_SLOT_DISPFRAME capabilities. To start the child process, invoke_dispatcher() was used.

To test our process management and spawning functionality, we wrote an additional test, spawn_one_with_capabilities(). This test spawned a process like normal, but also allocated a frame and passed a capability to it to the child. We modified the hello executable to see if it was able to access this passed capability.

Milestone 4

Message Passing

Responsibilities and Results

This milestone was a concerted effort between all four of us taking turns driving with the others helping with ideas and checks along the way. Andrew did most of the work towards the end once we passed the deadline to implement message passing. We fully finished this milestone 3 days after it was due, but mostly made up for it by implementing the bonuses for passing large strings in a shared frame and performance measuring. Once implemented, the original tests and our tests passed on both QEMU and the board.

Design Process

To begin setting up message passing, the first thing we did was create the LMP endpoint in spawn_load_with_caps() and initialize the LMP channel there. We had some trouble understanding the book on what things specifically to add to the Hakefile for this milestone, but we eventually figured out we needed both memeater and rpcclient for the tests and added them for both QEMU and serverboot.

We also got the init process to create its endpoint for LMP, as we had not done this before. From this, we implemented a basic skeleton for barrelfish_init_onthread() so the new process could construct the LMP channel with init's well-known endpoint as the receiver. At the same time, in spawn_with_caps(), init will register_receive() and wait for the new process to respond. The new process will register a send along the new channel and wait for init's acknowledgement in a blocking wait. At this point, we weren't using register_receive() properly because we didn't fully understand how the closure system worked, leading us to just put NOP_CLOSUREs for our closures. This began to give us some trouble down the line. Additionally, we used the regular send function for sending along the LMP channel instead of register_send(). This led to an interesting-looking implementation for this first week or so of implementing.

Luckily, we quickly already made a receive handler for the constant looping receive which taught us about how we should be implementing our register_receive() functions. Once we realized how we should be using handlers and closures, we began one of the many refactors we tried throughout the length of this milestone. A main help in the refactor was replacing much of our LMP channel initialization code with lmp_chan_accept(); we also then added an LMP channel pointer and a receive handler function into our spawninfo struct, which allowed us to better abstract setting up IPC and receiving over LMP channels.

Then, we began to implement the AOS RPC abstraction. We created a simple aos_rpc struct and filled out the functions for AOS RPC initialization, which encapsulated much of the channel setup procedure, and we decided to route the memory server, terminal server, etc. to the init process for simplicity. At this point, we finished moving over to the AOS RPC implementation and tests began passing! We then cleaned up what we added to the spawninfo struct as it was unnecessary, and we added a generic send handler for our register send function calls, as well as an acknowledgement receive handler. Additionally, we made an equivalent generic receive handler and an acknowledgement send handler. Once we completed these handlers, we got acknowledgements to be successfully sent and received, giving us some confidence in our implementation which wouldn't last long.

Once we knew the sending and receiving worked, we began working on specific messages, starting with numbers. We created a number payload, which was just a struct with an integer and the aos_rpc struct. We paired this with a send_num_handler, which gave us the ability to send numbers, but was inconsistent as our handlers didn't have a way to differentiate between different messages.

To improve our message handling and consistency of receiving messages, we assigned a number for each different message we wanted to send. We added this into our send handlers and this made our generic receive handler effectively a big if-else statement. With this more modular code, we implemented string message passing as well. We implemented string passing by taking advantage of the passed frame through the RPC channels, where the generic receive handler gets a string message case and copies out the string from the received frame (capref). Once it's copied out, we send the acknowledgement back so the sender can stop blocking.

Once we got strings and numbers to send, we tried to implement some more handlers, but we noticed that our handlers were in the wrong place (init.c) so we moved them to aos_rpc.c, since it made more sense logically. However, this caused even more problems due to some include statements not being linked properly since these handlers were also called in main.c. We eventually just moved some handlers to main.c to fix the problem, and it appeared that this is also what some other groups ended up doing.

In a regrettable move, we decided to "simplify" some of our blocking code to no longer check for transience and instead just abort on error since we weren't running into those loops. We then did this in our send handlers as well, but this effectively destroyed our blocking, so instead of a blocking while loop, we figured out that we could used exactly two event_dispatch() calls instead of waiting, since it was enough to allow the other process to receive our message and send an acknowledgement back. In hindsight, this wasn't a very good idea and likely had the possibility for many errors and issues.

Moving forward, we then added the putchar message with a send putchar handler that effectively worked the same as our send number handler, since they both only send primitive data. We then gave it a message type and added a case for it in the generic receive handler. Since it was pretty simple, it passed the tests to check it quickly.

At this point, the code didn't look too nice or seem very readable, so we refactored again. This mainly consisted of changing the generic receive handler to a switch case statement and adding an enum message type for each message, which also made it clear which messages were being sent and received. This also made it much easier to add new messages without conflicting with other group members taking up that number that a message non-intuitively corresponds to.

After a successful small refactor, we implemented the spawn with command line message along with its acknowledgement, which had to be different since this acknowledgement had to send data, specifically a PID, back to the sender. This first would require us to implement aos_rpc_spawn_with_caps() to allow us to send capabilities over the LMP channels. This function was structured nearly identically to how we constructed frames for passing strings, except with those two event_dispatch() calls and a receive to the PID returned from the PID acknowledgement.

Once we got the hang of this flow, we quickly implemented getchar. This was followed by our implementation of getting RAM from the memory server, for which we also used similar techniques but needed a new type of payload struct for returning a capref and a number of bytes allocated in it. The receive handler side was relatively similar to the others, this time giving the payload response data accordingly. However, we began to have trouble with page faults and memory problems. Funny enough, it was because we forgot to free the malloced payloads we were passing to the init process, so it was a relatively easy fix.

Now, for replacing the terminal read and write with our putchar and getchar, we had some serious trouble. Surprisingly, getchar seemed to work just fine relatively quickly, but putchar immediately caused an infinite loop when printing anything to the terminal. After a long time of digging into where it happened that was made much longer because gdb clearly wouldn't work in this context, we realized that we were almost exclusively using printf() for our debugging statements. This made printf() execute while trying to run putchar, which evidently would run putchar again and so on. Obviously, we shouldn't have been using them in the first place, but now we actually had to replace them all with debug_printf(). Luckily, this did fix the issue and it was mostly working, but we definitely missed a few printf() calls, as errors returned from our older code still had some printf() calls that would be further uncovered when we tried to break things.

At this point, we felt that our code was very confusing and it was incredibly difficult to debug, especially because of our reliance on those event_dispatch() calls. This caused us to temporarily branch off into a test branch in our repository and try to completely refactor our messaging code from the ground up, knowing what we knew now. Because it was much easier to write and understand, we caught up with our main branch's functionality quickly. The test branch's code focused on abstracting away as much of the sending, receiving, and blocking as possible, since this was the main cause of concern for our main code, and it was also something that we practically rewrote every time in main. Once we caught up with main, we ran into an issue with grabbing wrong messages and not re-adding them to the frame. At this point, we split our group up, where one tried to get main to work while the others worked on debugging this issue. Ultimately and maybe unfortunately, we got main to work first and continued with that branch. Even to this day, you can see our test branch in our repository frozen in time with much cooler code (unbiased opinion). However, we did eventually change over some of this code and use it in our main branch.

After that fiasco, we continued on with implementing getting all PIDs, waiting, exiting, and spawning with caps through the LMP channel. At this point, we were past the due date for this milestone, so we decided to start implementing bonuses to help us recover the points we were losing. We implemented a quick performance measurement function to measure the microseconds it took for each RPC call to be run during the tests.

Then, to finish this milestone off, we added some tests for getchar to make sure it wasn't as bad as putchar. After this, we were now confident that our code worked as expected and we demoed our code a few days later. Now, we could finally start on Milestone 5.

Milestone 5

Multicore

Responsibilities and Results

For this milestone, Peter drove for the first half and Daniel drove for the second half, with David proposing bug fixes and helping with design throughout. By the time our code was evaluated, we were able to boot up all four cores and run the grading script on each of them. However, our code was not fully complete as passing the module strings to the newly-booted core was implemented but not working. This caused the cores to boot correctly on QEMU but to fail on the actual hardware. A few hours after we were graded, we were able to fix the bug and get our code working correctly on both QEMU and the Toradex board.

Design Process

Because we were unable to complete the previous milestone by the due date and thus had to take time out of the week we were given for this milestone, our design decisions for this milestone were largely influenced by the time it would take to get a working implementation rather than by the elegance or performance of the solution. This shows when one looks at our code, which is hard to follow and redundant in some places.

Our first order of business was to get the other core to boot and then immediately crash. To do so, we had to set up quite a bit. Luckily, the instructions for the milestone were helpful and, even better, there was an Ed post laying out pretty much everything we had to do to get the boot driver and the CPU driver loaded.

Our first design decision regarded how we would allocate memory for the other cores. The book described two options: we could either split up the RAM capabilities that were given to our initial kernel and reserve some of them for the other cores, or we could simply allocate some of the memory from our physical memory allocator on the initial core. We chose the latter option because it involved just a single function call (in this case, we chose to allocate 256MB of memory for each core to distribute to its running programs as it sees fit) rather than requiring us to revisit the code we used to set up the memory allocator. By allocating it on the initial core, this memory was effectively reserved for the core we were bringing up, meaning we were free to forge a capability to it. If we were to make changes to this section of our code, we would likely implement a scheme for determining the total amount of memory we were initially given as well as the number of cores on the board so that we could dynamically split up the memory and get rid of the hardcoded 256MB value.

Next, we had to set up quite a few memory regions for things like the KCB, the core_data struct, the kernel stack, and so on. Because we wanted to do things incrementally while testing whether or not they worked, we chose to allocate these all in separate frames. Looking back, this greatly lengthened our code and the same effect could have been achieved if we had simply allocated a large region of memory capable of holding all the smaller memory regions required for the process to boot and then split that up using some kind of struct or macro. Regardless, what we did eventually worked.

To start the core itself, we simply followed the instructions in the milestone description (for creating the memory regions) and the Ed post (for loading the boot driver and CPU driver binaries). Unfortunately, we got stuck on a bug that cost us over a day of debugging: we were able to boot the second core and load the init binary, but we were running into a panic immediately after. Upon making an Ed post, it was suggested that our CPU driver was not properly relocated. Indeed, adding the kernel offset when relocating the CPU driver allowed init to be started correctly.

From here, we needed to get process management and memory allocation working on the second core. To do this, we would need to pass information about the available memory to init. The book described two approaches to solving this problem: we could either pass the available memory regions through command line arguments, or we could pass them through the shared memory of the URPC frame. We chose the latter. To start, we added the 256MB memory region to a bootinfo struct that we initialized for the new core. Right after the bootinfo struct, we also passed other information to the core like the module strings. Once this all was in place, we mapped the bootinfo struct and the additional parameters into our memory when the booting core was up. To make sure we had access to the memory we passed from the initial core, we forged capabilities to the memory we were given and initialized the local memory manager with those capabilities. Again, this memory was already reserved in the initial core's memory manager, meaning that forging these capabilities was a safe operation.

For this milestone, we simply threw away the address to the URPC buffer once we were done using it to start a core. In the next milestone, we found that we needed to record its location so that we could use it for communication.

Once we had all of this done, our grading code, init, memory management, and process spawning was all working on the second core. Once we modified the code in the BSP's init to boot the other cores, we were pleasantly surprised to find that the other cores functioned as well. From here, the rest of the milestone was a matter of implementing the simple coreboot bookkeeping functions and fixing the bug with our module strings not being properly passed, which turned out to be a simple issue with the loop we were using.

Milestone 6

User-Level Message Passing

Responsibilities and Results

For this milestone, David and Peter took turns driving, with Daniel helping with design. By the time our code was evaluated, we were able to boot up two of the cores and run the grading script on each of them, and were shown to successfully pass spawn process messages in between and send PID acks back to the original core. However, our code was not fully "complete" as it was difficult for us to do this between all four of our cores. Later on, David and Peter got it to work between all four cores and fully test it before starting on Milestone 7.

Design Process

For the beginning of the milestone, we started by making init's event dispatching non-blocking and yielding the thread once we got an event. We did this with a while loop that polled the error returned from even_dispatch_non_block() until it wasn't LIB_ERR_NO_EVENT. We planned to remove the while loop later to allow for alternating polling between event dispatching and UMP message receiving.

Then, we set up six URPC frames, so cores 1-3 could communicate with core 0, with one frame handling the core to monitor direction and the other frame handling the monitor to core direction for each non-BSP core. We also created a ump_chan struct as a circular buffer for each of the URPC frames. Alongside this, we created a buffer init function and enqueue and dequeue functions. We then initialized all these frames in bsp_main() and prepared our spawn with command line function to handle different cores.

Once we got the skeleton of the UMP interface, we moved the UMP structs to shared memory and increased the size of the UMP frames from one page to three. With this setup, we were able to send a message to another core by spawning the hello program. We were still having trouble sending acknowledgements at this point, though.

We improved our testing by trying to spawn alloc restricting which cores ran certain parts of the tests. From here, we used an enum to type our messages (either SPAWN_CMDLINE or ACK, for now), which led us to poll for and identify specific UMP messages. We then built out methods for the cases for receiving these messages. However, this didn't yet allow us to send acknowledgements successfully. We thought this might have been due to caching, so we added the dmb() statements in the appropriate places, but it seemed to do nothing. We assume this is because we were running these tests only on QEMU up to this point.

After this, we were debugging when we realized that there were a lot of places in our code where we weren't previously error checking from many functions' returns, so we added more error checking to make debugging easier in the future.

To continue on getting acknowledgements to work, we discovered that our UMP payload struct that we used to pass data through the circular buffer didn't have send and receive core fields, and instead only had a slot for a receive core. This made it impossible for the receiver core to know where to send the acknowledgement, so we added another field to differentiate the two, which also allowed us to send to any of the four cores by passing the message through core 0 and having BSP forward messages that it's not meant to receive. We then added a method to print out our circular buffer to help us debug how our messaging flow was working, which helped us realize that messages were piling up in the buffer and not being dequeued. This made us realize that we had tried to receive the wrong message in certain spots of the buffer, since we made our dequeue function only dequeue at a fixed spot (the tail) until it was moved up. We had made it so that if a message was of the wrong type, it would just stall since the buffer would not be able to dequeue anything further up the queue. To deal with this issue, we changed the circular buffer into a list, where we would loop through all 64 spots linearly from the start, dequeueing the first message that matched what we were looking for.

At this point, our code was working enough to pass the tests we made, but we planned on improving the implementation and readability of the code since we were worried it would cause issues in the future, especially since the linear searching on the buffer could possibly grab the wrong message in some cases and mess everything up. We quickly implemented the fragmentation bonus. We allowed a payload to be a maximum of 128 bytes instead of the 58 bytes that would fit in the cache line of the buffer slot with metadata. We did this by adding a fragment number and total fragments fields to our cache lines in the buffer. This allowed us to check if we should immediately dequeue more than one fragment at a time from the buffer. Similarly, we calculated the size of the payload we enqueue and broke it up into cache-line-sized messages on the queue in order. We tested this by sending a command line spawn message to another core with arguments longer than 58 characters. In this case we sent a hello spawn request with a long string and it worked.

After this, we refactored our buffer and made it a circular queue again, which greatly simplified our enqueue and dequeue functions and removed most of the worries about getting the wrong message in polling. This made our buffer function practically identical to how it's shown in Figure 8.6.

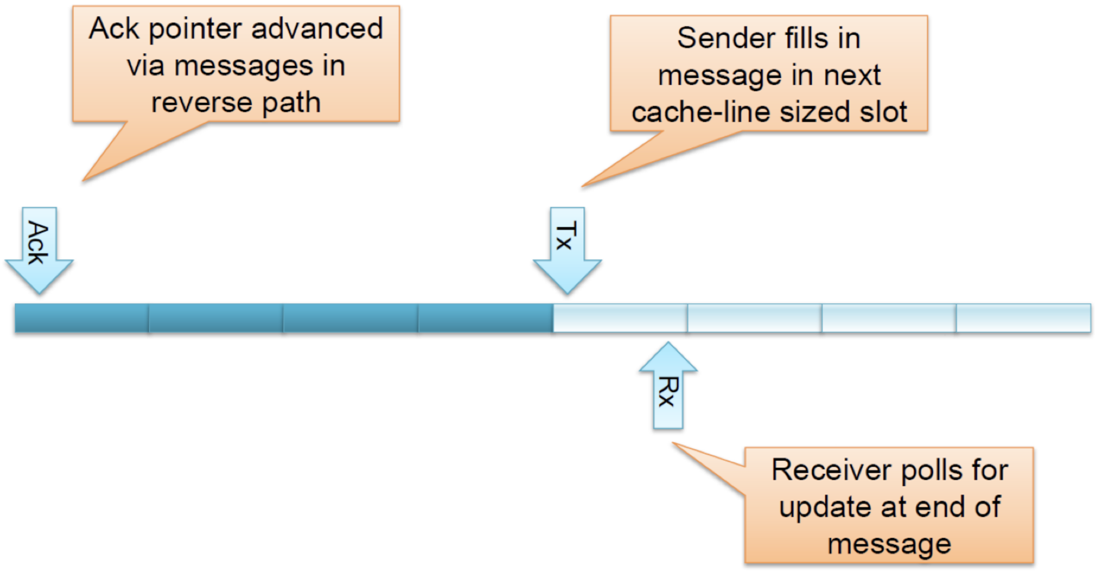

Ack pointer advanced via messages in reverse path
Sender fills in message in next cache-line sized slot
Ack
Tx
Rx
Receiver polls for update at end of message

Figure 8.6: UMP message channel

We also added more strict tests which confirmed that we could concurrently send many different messages and their acknowledgements across our URPC frames, even with fragmentation, and not have any messaging issues.

Once we were confident that our implementation was logically sound, we began stress testing how many processes we could spawn on different cores. However, once we tried this, we ran into a memory error. We investigated this and discovered that our vnode_map() function was panicking. Luckily, the panic told us to simply increase the macro ARMV8_CORE_DATA_PAGES slightly and it started working again. We still think we might have just kicked the can down the road, though, and it's possible that we spawned too many alloc processes.

Milestone 7

Shell

Responsibilities and Results

David began this milestone by attempting to fix the issue with our vspace logic that seemed to be cropping up. After a few days of work, he found a workaround that allowed the development of our shell to begin in earnest. Peter and Daniel then implemented the userspace serial driver and the various shell commands. By the end of the milestone, we had a very useable albeit very unstable shell capable of process management, filesystem manipulation, and more. The instability arose from the aforementioned vspace issue. The only missing feature at the moment is that processes spawned on other cores are not able to print or receive characters. This is because we have not implemented a UMP call for sending characters across cores. Although we did not implement any bonuses, we did fix a bug in the provided code and we added a few useful commands not listed in the book, such as “pwd,” “lsmod,” “cd,” and “touch.”

Design Process

Our first step after finding the workaround for the vspace issue was to get the userspace terminal driver working. Luckily, we had already implemented getchar functionality over LMP, meaning all we had to do was to find and map the registers for the UART. We set things up so that the system type (QEMU or IMX8X) was automatically detected and the correct driver address and commands would be used regardless of the platform. For simplicity, we decided to run the terminal driver within init, meaning we only had to map the DevFrame into our regular address space. Although we could have run the terminal server in its own process, we decided that this approach was more in line with how we constructed the rest of the operating system.

The next step was to start implementing the shell. We chose to run the shell as a separate process so that there weren’t any conflicts with the message dispatch loop. Our shell program effectively loops over and over again, asking for commands, transforming them into tokens, and executing a command based on the first token. The first command we implemented was “echo,” allowing us to perfect our tokenizer. In particular, we programmed our tokenizer such that, when it encounters a quote, it considers the rest of the text until the next quote as a single token. As an added optimization, we delineated the tokens by simply transforming the space or quote at the end of a token into a null terminator. This way, we didn’t have to copy the string or manage an array of strings in order to tokenize the command.

With the basic algorithm down, the implementation of the rest of the commands was fairly trivial, consisting mostly of calling the functions we had implemented throughout the quarter. At one point, we did run into an issue with our wait() function, which was blocking LMP messages (like the one that signals when a process exits) while waiting for a process to exit. We fixed this by waiting at the API level (that is, we made the actual message itself nonblocking). To achieve this, we returned the exit code if the process had exited or a special value if the process had not yet exited. We also used this same strategy to return errors when spawning the process (that is, we returned the PID if the process spawned successfully or a special code if not). Given more time, this would be better implemented using a struct containing both a PID (or exit code) and a boolean indicating whether or not the process had spawned (or exited) successfully, as this would free up all PIDs and error codes for use by the process manager (our current implementation only allows PIDs and error codes less than the special return value).

The implementation of the filesystem was also straightforward. We chose to initialize the system with a single file (created in the RAMFS initialization method) so that we could test the output of the "cat" command. The biggest obstacle as far as the filesystem went was the lack of comments or documentation in the RAMFS files. As a side note, we found and fixed a bug in the RAMFS that we think may be eligible for the ongoing bonus point challenge. Specifically, we switched the order lines 647 and 648 in ramfs.c. Originally, the goto was placed before the error was set, causing the error condition to return an uninitialized value. Switching the lines caused the proper error to be returned.

Overall System Design Summary

When it comes to overall system design, the book seemed to consistently provide two alternatives: we could either create a separate domain for a service (like a memory server) or we could simply run the service within init. When faced with this choice, our group always chose to run the service in init. As such, init takes care of everything LMP and UMP related, including forwarding messages on to other cores.

Although the decisions we made resulted in simplicity, that simplicity was achieved at the expense of modularity and performance. Of course, having init serve requests of all different types means that init becomes a massive program, making it difficult to maintain and understand. Furthermore, many of the same bottlenecks that one would find in an operating system with a monolithic kernel are present in our system as a result of our decisions. For example, our init process cannot serve LMP and UMP requests at the same time. Instead, LMP and UMP messages are processed in turn, resulting in potential delays or wasted processor cycles when one service is used more than the other. Furthermore, a call to one of these services that takes any appreciable amount of time will result in delay for services of any type provided by init.

Despite these shortcomings, we feel that putting all services in init was the correct decision given the difficulties we faced in coping with the workload of the class. Although a cleaner and more performant solution was possible, having init run all of our services saved us time. If we were to take the project further, we would likely attempt to run some of these services in their own domains to truly leverage the operating system’s architecture.

Appendix

Shell User Guide

After the system has started and any grading tests have run, a separate process containing the shell is run, prompting the user with a dollar sign (`$`). From here, the user may enter a command, which is then tokenized by the interpreter and executed. By default, tokens are delineated by spaces. If the user wishes to create a token that includes a space, they may surround the token in double quotes as follows:

```
$ echo “hello world”
hello world
$
```

A number of built-in commands exist for miscellaneous purposes. A list of miscellaneous commands and their functions follows:

`echo [string]`	Prints the first token
`run_memtest [size]`	Allocates a block of the provided size (4096 bytes if no size is provided), writes to each byte, and reads from each byte, generating an error if the incorrect value is read
`lsmod`	Prints a list of ELF modules included in the boot image
`time [cmd]`	Times the provided command (formatted normally), printing the duration of the command’s execution in milliseconds after the command completes
`help`	Prints a list of commands organized by general function

Furthermore, the shell provides facilities for process management. A list of process management commands follows:

`run [cmdline] [&]`	Runs the provided program (if more than one token is provided after the “run” token, the following tokens are treated as command line arguments) either in the foreground (if no “&” is provided) or the background (if “&” is provided after the command line)

`oncore [coreid] [cmdline] [&]`	Similar to run (see above) but with the added ability to specify the core on which the process will be run (note that oncore is not currently supported—processes that attempt to print while running on an APP core will cause a segmentation fault as no UMP service currently exists for the terminal driver)
`ps`	Prints a list of running processes (including PID and name)

Note that no “kill” command is provided and some programs (such as alloc) do not terminate, meaning that starting these programs in the foreground will render the shell inoperable until the system is rebooted.

Finally, the shell has a suite of file management commands capable of manipulating the provided RAMFS. Upon boot, the RAMFS is initialized with a single file named “test.txt” in order to demonstrate the functionality of the “cat” command. A list of file manipulation commands follows:

`pwd`	Prints the current working directory
`ls`	Prints a list of files in the current directory, including information about the type, size, and name
`touch [file]`	Creates an empty file in the current directory with the specified name
`rm [file]`	Deletes the file with the specified name in the current directory (if it exists)
`cat [file]`	Prints the contents of the file with the specified name in the current directory (if it exists)
`mkdir [dir]`	Creates a directory with the specified name in the current directory
`rmdir [dir]`	Removes the directory with the specified name in the current directory (if it exists)
`cd [dir]`	Changes the current working directory to the directory at the specified absolute path

Note that all commands except for “cd” use relative paths. The use of cd is illustrated in the following example:

```
$ pwd
/
$ mkdir example
$ cd /example
$ pwd
/example/
$
```

Note that the root directory is specified as `/`.

As a final note, our operating system is currently unstable due to a problem with our vspace logic. We’ve been trying to track it down, but haven’t had success despite our efforts. As a result, the shell often crashes our operating system when executing commands. When this happens, the only option is to reboot.