Advanced Operating Systems

Report

Group B - Fall 2017 - DoritOS

Carl Friess, Sven Knobloch, Sebastian Winberg

Memory Management

We chose a linked list as our fundamental data structure. Each node represents an allocated or free area of RAM and the nodes are stored in the same order as the corresponding regions occur in memory. This allows us to easily and efficiently coalesce adjacent regions.

```
struct mmnode {
    enum nodetype type;     ///< Type of `this` node.
    struct capinfo cap;     ///< Cap in which this region exists
    struct mmnode *prev;    ///< Previous node in the list.
    struct mmnode *next;    ///< Next node in the list.
    genpaddr_t base;        ///< Base address of this region
    gensize_t size;         ///< Size of this free region in cap
};
```

A memory region node

The struct above describes each node in the linked list. The field `type` marks nodes as free or allocated while the `base` and `size` fields define the precise memory region.

In contrast to the other fields, the `cap` field does not directly relate to the memory region described by the node. Instead it is the RAM capability for the RAM region, which was initially added to the memory manager. As we will see, this is necessary in order to later release allocated memory.

Looking at the `cap` field in more detail, it actually consists of a struct storing not only the capability for the initial region, but also the information describing it:

```
struct capinfo {
    struct capref cap;
    genpaddr_t base;
    gensize_t size;
};
```

Struct storing a RAM region and it's capability

The `prev` and `next` fields are used for traversal. We considered using the `collections/list` library but decided against it as we need specialised operations on the list based on the data stored in each node. Ultimately it seemed simpler to manually implement the linked list. In hindsight it might have been wise to instead somehow modify the library to better suit our needs, since it turns out that we used linked lists for many similar purposes later on.

Adding RAM regions to the memory manager

Although in practice not used, the memory manager allows for an arbitrary amount of memory regions to be added to its purview by calling `mm_add(struct mm *mm, struct capref cap, genpaddr_t base, size_t size)`. For each of these regions it receives a RAM capability spanning the entire region.

To add a region, the memory manager creates a new node, marks it as free and stores the RAM capability together with the other relevant information. The node is then added to the linked list in no particular order. As we will see, it is only necessary for nodes of the same region to be correctly ordered and grouped together.

Allocating memory

When servicing a request for a RAM capability of a given size, the list is traversed and a node in the list is chosen by first fit. The requested size and alignment are checked to be a multiple of the page size (`BASE_PAGE_SIZE`) and rounded up accordingly. Other sizes and alignments are not useful, since they cannot be mapped anyway.

Once a node is found it is checked for alignment and the necessary padding is calculated to meet the alignment criteria. If the size of the region is still sufficient, it is then split into two nodes at the front of the region. The resulting node is split again at the back if the remaining memory region is still larger than the requested size. Like this an allocation can cause a node to become up to three nodes.

This splitting will cause some overhead when traversing the linked list. However, it is only rarely necessary to split at the beginning of a RAM region to meet alignment criteria. Additionally, we are eagerly coalescing nodes when releasing memory, which should help keep the overhead low.

Once the final node has been created the initial RAM capability is retyped and returned to the client. The new capability is not stored, since the client must return it when releasing the memory.

Releasing memory

Clients of the memory allocator can return memory capabilities using the `mm_free(struct mm *mm, struct capref cap, genpaddr_t base, gensize_t size)` function. The manager then uses the `base` argument to find the corresponding node in the linked list. Once found, the returned capability is simply deleted, allowing the initial RAM capability to be retyped again for the same region.

Next, the next and previous nodes in the list are checked for the possibility of coalescing. We found that unifying coalescing algorithm of both the preceding and subsequent nodes simplified some otherwise complicated case distinctions, leading to this function for coalescing:

```
void coalesce_next(struct mm *mm, struct mmnode *node) {
    // Sanity checks
    assert(node->next != NULL);
    assert(node->base + node->size == node->next->base);

    // Update size
    node->size += node->next->size;

    struct mmnode *next_node = node->next;

    // Remove the next node from the linked list
    node->next = node->next->next;
    if (node->next != NULL) {
        node->next->prev = node;
    }
```

```
        // Free the memory for the removed node
        slab_free(&mm->slabs, next_node);
    }
```

Algorithm for coalescing RAM region nodes

Unfortunately, it is not possible to merge RAM capabilities. This is another reason, beyond simplicity, why we only store the initial RAM capability for the entire region and then retype it for each allocation.

Gathering metadata

We added a function (`mm_available(struct mm *mm, gensize_t *available, gensize_t *total)`) to determine the available and total memory managed by the memory manager. This proved very useful in testing for memory leaks and other bugs.

Since the memory manager currently does not keep any such data, the function needs to traverse the entire linked list each time it is called, which is quite inefficient. However, we ended up only using it for debugging and so deemed this implementation acceptable.

Allocating memory for the nodes

Obviously we also need memory to store each of the nodes in the linked list. We used a slab allocator for this purpose. However, even the slab allocator needs to request more memory form time to time.

We were provided with an implementation of a slab allocator, but added the implementation of `slab_refill_pages(struct slab_allocator *slabs, size_t bytes)`. The refill process is simple. It requests a RAM capability from the memory manager, retypes it to a frame capability and maps it.

Obviously this creates a circular dependancy on the memory manager. We solved this issue by determining that the slab allocator should always have 5 or more free slabs for any request to succeed. When this lower bound is crossed during a RAM allocation, the refill is prematurely triggered. A flag in the state of the memory manager is set to prevent an infinite refill loop.

Allocating slots for capabilities

Thanks to the provided `twolevel_slot_alloc` and `single_slot_alloc` slot allocator implementations, the allocation and refilling of slots was not particularly problematic and very little pre-emptive refilling was necessary.

We were also provided with a slot pre-allocator. However, we weren't sure what was intended for and didn't use it. Instead we simply implemented the `slab_refill_no_pagefault(struct slab_allocator *slabs, struct capref frame, size_t minbytes)` method using only `slab_default_refill()`, which executes the previously discussed `slab_refill_pages()`.

Spawning Processes

Spawning is the process of creating new dispatchers in order to create schedulable processes for the system. It is subdivided into several smaller, more managable steps to give an easier overview.

CSpace

We start by creating a CSpace for the child process. First thing is to create an L1 CNode and provide a L2 CNode to start storing the capabilities. Then the new dispatcher itself is initialized and the capability is copied over, as well as it being retyped to an Endpoint so that it has access to its own Endpoint. 3 new L2 CNodes are then created to provide the initial slots for the process while it is doing setup and can't allocate additional ones. Another new L2 CNode is then added to store some initial page-sized RAM capabilities. The entire CNode is filled to make sure the new process has plenty of memory to get started. Finally, one last CNode is allocated so that there is room for the initial `paging_state` to be copied over at a well known location. One other thing of note is that the IRQ capability is also copied over. It is given to all processes, which is often unnecessary but also saves an additional RPC call to get the capability when children do need to register IRQs.

LMP

To allow IPC, the new process must have some way to communicate with `init`. First, the capability for `init`'s endpoint is copied over to a well known location. This means the new process can send to `init`'s endpoint at any time. Since the child's endpoint does not exist yet, `init` listens to all incoming messages and waits on a registration message. This message will then hold the capability to the child's endpoint which `init` can then use to finalize the bi-directional channel.

VSpace

The VSpace is used to pass the `paging_state`. This entails allocating some memory for it as well as some initial slab allocators for it to use. More about this is discussed in the `Virtual Memory and Paging` section.

ELF

Spawning the elf is done with the provided elf library. The first step is to load the elf, which uses the `elf_allocator_callback`. This function lets the `elf_load()` call allocate the memory it needs to load the elf data. Then we find and store the location of `.got` section so we can later notify the dispatcher about it.

Dispatcher

First thing for the dispatcher creation is to allocate memory for it. After that, we can assign the various fields of the DCB to their predefined values and additionally assign some of the address locations from previous sections, like the `.got`.

Arguments

Another crucial part of spawning processes is passing the arguments. For this, a frame is allocated where the parsed arguments are written to. The capability to this frame is then copied to the child's

CSpace so it can call up the arguments. One thing to note here is that the address of the arguments has to be saved into register `r0` in order for it to work properly when booting the process.

Spawn and Process List

```
struct process_info {
    struct process_info *next;
    domainid_t pid;
    coreid_t core_id;
    char *name;
    struct capref *dispatcher_cap;
    struct lmp_chan *lc;
};
```

Process Info

There is only one thing left to do before spawning the process and thats bookkeeping. Before spawning, the process' `process_info` is added to the list of all processes. Keeping track of processes allows for some crucial functionality such as arbitrary binding that comes into play later in the system. Last step is to invoke the dispatcher and we are off to the races!

Cleanup

Final bit of `init`'s side of spawning is doing some cleanup. This includes unmapping unneeded frames that were mapped to write data like paging state and such to the child as well as freeing the slots saved in the spawninfo.

Initialization

Once the process is actually up and running, it needs to initialize itself. Just like `init` it will run through the initialization, including `paging_init`, `ram_alloc_init`, etc. but will additionally initialize LMP and AOS RPC. To initialize LMP, the process creates a new LMP channel and sends a registration message to `init`, along with it's endpoint capability. In the spawning process `init` registered itself to listen to incoming LMP messages and will therefore receive this message, register the process' endpoint and respond, finalizing the initialization. Only thing left is to initialize AOS RPC and then the process can enter `main`.

Lightweight Message Passing (LMP)

Lightweight Message Passing is a key protocol in establishing inter-process communication. With the involvement of the scheduler, it allows fundamental functionalities to be implemented efficiently. In particular, it is the only protocol in our system, which allows capabilities to be natively sent between processes.

Communication protocol

LMP communication runs over channels. We will be using the same channels to send many different messages and therefore need a uniform protocol to allow effective networking. The complete protocol specification is listed in `include/aos/lmp.h`.

To simplify the distinction of message types we decided to dedicate the first argument of every LMP message to the message type (described by `enum lmp_request_type`). The following arguments are defined based on the message type. In some cases we use two consecutive messages to make use of general protocol implementations, such as `lmp_send_string()` and `lmp_recv_string()`. In the case of LMP channels used in the URPC API, the first argument is composed of the standard message type (`enum lmp_request_type`) and the 6 bit URPC message type.

AOS RPC

We used LMP to implement (almost) the full set of `aos_rpc` functions. We decided to consolidate the memory manager, spawn server and process management in the init process, while the terminal server is later removed from init. We therefore absolutely need an LMP channel with init, which is constructed during spawning.

When a process is spawned, init also registers an LMP request handler function in a closure with the LMP channel on the default waitset. The handler is called for all incoming messages from the process on init. Thanks to the closure the handler receives a reference to the LMP channel. After being called the handler reregisters itself to receive future messages. A reference to the LMP channel established during spawning is also stored in process management, so an LMP channel can be used to retrieve further information about a process.

To service AOS RPC requests, the first step in the handler function is a switch statement to differentiate between the request types based on the message type sent in the first argument. Init then executes the necessary steps to (hopefully) satisfy the request and sends the appropriate reply. On the other side, the client process waits for a response using a blocking call to `lmp_client_recv()`, which is implemented using the default waitset similarly to how init waits for incoming messages.

```c
// Blocking call for receiving messages
void lmp_client_recv(struct lmp_chan *lc, struct capref *cap,
                     struct lmp_recv_msg *msg) {

    // Use default waitset
    lmp_client_recv_waitset(lc, cap, msg, get_default_waitset());

}

// Blocking call for receiving messages on a specific waitset
void lmp_client_recv_waitset(struct lmp_chan *lc, struct capref *cap,
                             struct lmp_recv_msg *msg, struct waitset *ws) {

    int done = 0;
    errval_t err;

    err = lmp_chan_register_recv(lc, ws, MKCLOSURE(lmp_client_wait, &done));
    if (err_is_fail(err)) {
        debug_printf("%s\n", err_getstring(err));
        return;
    }

    while (!done) {
        event_dispatch(ws);
    }

    err = lmp_chan_recv(lc, msg, cap);
    if (err_is_fail(err)) {
        debug_printf("%s\n", err_getstring(err));
    }

}
void lmp_client_wait(void *arg) {
    *(int *)arg = 1;
}
```

Blocking LMP receive implementation

An important assumption is made here. Namely, that init only handles one request at a time and requests do not interleave. This assumption holds true with one exception: RAM capability requests. Unlike all other RPC calls, these requests can occur at any time due to demand paging. On init's side this isn't an issue. Init will finish treating the initial request and then service the memory request. However, the client process might receive a completely different message when waiting for the reply to the memory request.

We fixed this issue by checking for the message type of the received response and requesting a resend from init using the message type `LMP_RequestType_Echo`. A cleaner implementation would probably be to store the received message in the client until the memory request has been handled. However, in our case this proved to be a much simpler solution.

```
// [ ... ]

// Initializing message
struct lmp_recv_msg msg = LMP_RECV_MSG_INIT;

do {

    // Receive the response
    lmp_client_recv_waitset(chan->lc, retcap, &msg, &chan->mem_ws);

    // Check if we got the message we wanted
    if (msg.words[0] != LMP_RequestType_MemoryAlloc) {

        // Allocate a new slot if necessary
        if (!capref_is_null(*retcap)) {
            err = lmp_chan_alloc_recv_slot(chan->lc);
            if (err_is_fail(err)) {
                debug_printf("%s\n", err_getstring(err));
                return err;
            }
        }

        // Request resend
        err = lmp_chan_send9(chan->lc, LMP_SEND_FLAGS_DEFAULT, *retcap, LMP_Reques
        if (err_is_fail(err)) {
            debug_printf("%s\n", err_getstring(err));
            return err;
        }

    }

} while (msg.words[0] != LMP_RequestType_MemoryAlloc);

// [ ... ]
```

Receive sequence for LMP RAM request responses

We also use a separate waitset (`chan->mem_ws` above) to wait on memory request responses. This is due to issues with some sequences of RPC calls causing the responses to be delivered to the wrong handler function.

Sending strings and buffers

Beyond simple RPC requests, we decided to implement a framework for sending more complex data. We also decided we wanted to support "zero-copy" sending of strings (and later buffers). This framework powers AOS RPC calls such as `aos_rpc_send_string()`, `aos_rpc_process_get_all_pids()` and `aos_rpc_process_spawn()`. We came up with a system of functions with heretical dependancies to making it easy to implement high-level functionality easily.

```
                               lmp_send_string()
                               /               \
             lmp_send_short_buf()             lmp_send_frame()


                               lmp_recv_string()
                                       |
lmp_recv_short_buf()      lmp_recv_string_from_msg()         lmp_recv_frame()
            \                 /                \                   /
   lmp_recv_short_buf_from_msg()              lmp_recv_frame_from_msg()
```

Framework for sending buffers and strings

The `*_short_buf*` family of functions sends short buffers (up to 28 bytes) by encoding them in the arguments of an LMP message. The `*_frame*` family on the other hand use frame capabilities and can be used for "zero-copy" sending. The functions with the suffixes `_from_msg` take a previously received message and handle it, whereas those without first blockingly receive an LMP message and pass it to their counterparts with the suffix. The receive functions `lmp_recv_short_buf_from_msg()` and `lmp_recv_frame_from_msg()` both send acknowledgments and their sending counterparts wait for these using `lmp_client_recv()`.

The high-level functions `lmp_send_string()`, `lmp_recv_string()` and `lmp_recv_string_from_msg()` are designed to automatically switch between the short and long buffer implementations based on the length of the string. This framework made the implementation of many RPC calls more uniform and intuitive.

We also implemented a second set of functions for the URPC API (discussed later). These do not use acknowledgements and are designed specifically for generic buffers. This decision mainly comes down to performance reasons. The acknowledgments significantly slow down the protocol and are usually (when necessary) already part of the application layer built on the URPC API.

```
                            lmp_send_buffer()
                            /               \
         lmp_send_short_buf_fast()               lmp_send_frame_fast()


                            lmp_recv_buffer()
                                   |
lmp_recv_short_buf_fast()  lmp_recv_buffer_from_msg()   lmp_recv_frame_fast()
             \                /                \                  /
   lmp_recv_short_buf_from_msg_fast()       lmp_recv_frame_from_msg_fast()
```

Framework implementation optimised for URPC

Virtual Memory and Paging

Initialising paging state

The paging state contains all the information about which ranges in the virtual address space are allocated and which page mappings (page table entries) have been made. Beyond this, the paging state contains the state for two slab allocators, which are used to provide memory for the data structures containing the just mentioned information. A reference to a slot allocator is also stored for reasons we will discuss in the next section.

Since init spawns all processes, the initialisation of paging states (including its own) is always carried on init. This is necessary because init needs to set up the virtual memory of new processes to contain mappings to the executable code of the new process among other things. When initialising its own paging state, it uses a static buffer as initial storage for the slab allocators until RAM from the memory manager can be mapped and used. For child processes this is not necessary, since its own paging state is already set up, so `frame_alloc()` is used instead.

Initialising paging

During the startup sequence of every process (including init) the `paging_init()` method is called. This will either initialise the paging state in the case of init or import the previously set up paging state.

Next a temporary slot allocator is initialised, as we are about to allocate some memory for the exception handler's stack. The standard slot allocator requires page fault handling and paging to be working so we use this temporary slot allocator in the interim. We found that the slots in the `SLOT_ALLOC0` cNode are unused, so the temporary slot allocator just allocates these slots one at a time.

Once the stack for the exception handler is set up, the handler can be registered and now we can start using the default slot allocator.

Moving the paging state

As mentioned above, the paging state for new processes is initialised in init and needs to be moved or rather imported to the child process. Luckily we are using a slot allocator to allocate memory for the data structures describing the state. This means that we know exactly which memory is used to hold these data structures. By simply mapping the appropriate frames to the same addresses in the child's address space the child can retain access to the data structures.

Now that the child can access the data, it just needs to know where it is. We solved this by just placing the `struct paging_state` at the well known address `VADDR_OFFSET`.

Finally we need to ensure that the child process can modify the vNodes created by init on its behalf. The following function recursively walks the paging state and copies the necessary capabilities into the child's CSpace. Here we also use the `SLOT_ALLOC0` cNode mentioned in the previous section.

```
static void spawn_recursive_child_l2_tree_walk(struct spawninfo *si,
                                               struct pt_cap_tree_node *node,
                                               int is_root) {

    static size_t next_slot = 0;

    // Reinitialise at root
    if (is_root) {
        next_slot = 0;
    }

    // Recurse to the left
    if (node->left != NULL) {
        spawn_recursive_child_l2_tree_walk(si, node->left, 0);
    }

    // Build next capref
    struct capref next_cap;
    next_cap.cnode = si->slot_alloc0_ref;
    next_cap.slot = next_slot++;

    // Copy the capability
    errval_t err = cap_copy(next_cap, node->cap);
    if (err_is_fail(err)) {
        debug_printf("spawn for %s: %s\n", si->binary_name, err_getstring(err));
    }
    assert(err_is_ok(err));
```

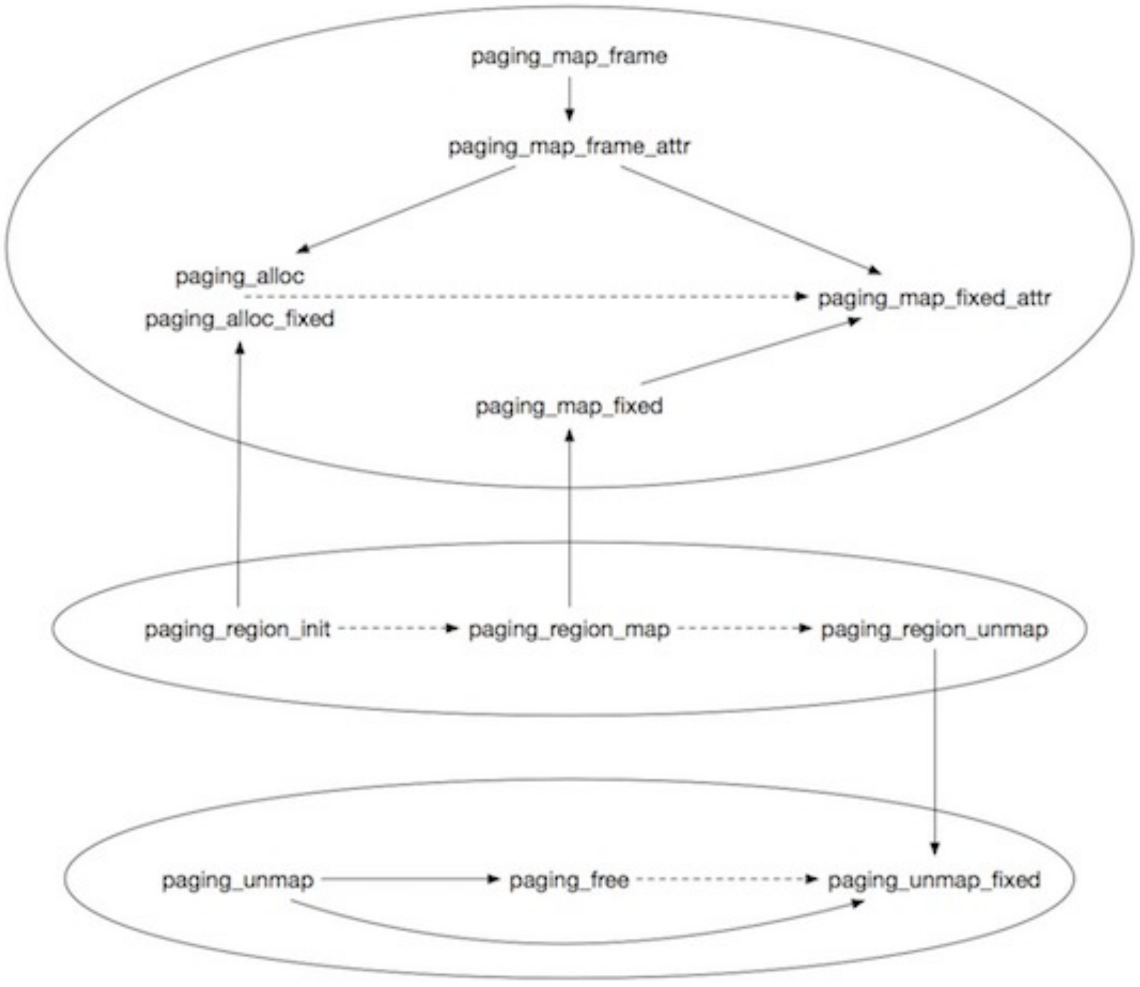
```
        // Mutate the capref to reference the child cspace
        next_cap.cnode.croot = CPTR_ROOTCN;
        node->cap = next_cap;

        // Recurse to the right
        if (node->right != NULL) {
            spawn_recursive_child_l2_tree_walk(si, node->right, 0);
        }

    }
```

Recursively copies vNode capabilities to child

After these steps have completed, the child can import the paging state by simply setting the current paging state reference to the well known address named above and setting the L1 vNode capability reference to the well known location of the first slot in `cnode_page`.

Discussion of general structure

For the structure in this project we intended to separate the vspace from the cspace. Therefore we introduced two different datastructures. A tree for the pagetable structure containing the mapping capabilities and an allocated and free list for the vspace.



Structure of virtual memory management

For the structure of the virtual address space we chose to use two list. An list of allocated regions and a list of free regions. Furthermore do we have the variable `free_vspace_base` to indicate that anything passed this address is free virtual address space.

Allocating virtual address space

Allocating a fixed region in the virtual address space uses the function `paging_alloc_fixed()`. We thereby allocate a fixed area in memory represented by the `struct vspace_node` and add it to the head of the allocated list. This function is usually only called shortly after initialization. `paging_alloc_fixed_commit()` has to be called right after it to restore a consistent pair of allocated and free list data structures. This function is in it's current implementation very inefficient. Though we created a more sophisticated and efficient version, we unfortunately could not change it, since we didn't have the time to properly test and verify that it works.

The more commonly used function `paging_alloc()` does not suffer from this problem, since it walks through the free list until it finds a big enough region and removes it from that list, with eager coalescing happening. If it does not find a big enough region, it just increments the `paging_alloc_fixed()`. After that we insert it to the allocated list, and return the base address of region.

For `paging_free()` we remove the vspace node from the allocated list and insert it in the sorted free list, where we eagerly coalesc.

Mapping

For the mapping we have implemented a two level tree structure with left, write and subtree pointers. The first level is thereby a binary three with l2 pagetable capability node. Each of these nodes has another subtree to keep track of their mapping capabilities. In `paging_map_fixed_attr()` we search for the first level of the tree to find a potential l2 capability. If it does not find one it creates the node and l2 pagetable capability. Next we allocate another node, add it to the subtree and map the frame to the appropriate slot in the l2 pagetable.

Furthermore did we implement `paging_unmap_fixed()` which walks the l2 tree until it finds the node which has the offset `l1_offset`. If it has reached this node it will move to the second subtree to find the actual mapping capability. After we have found the mapping capability we call `vnode_unmap()` to unmap it from the l2 pagetable, destroy it and free the slot.

Handling page faults

The process of handling page faults is somewhat trivial. Essentially the handler just requests a frame to fill the page which faulted and maps it. To support paging regions we also check if the address space where the fault occurred was previously allocated. If not, the address space is allocated to prevent attempts to map another frame into the same region.

We also check for `NULL` dereferencing and stack overflows, as well as rejecting page faults in the part of the address space reserved for the kernel.

Unfortunately, our handler uses `paging_alloc_fixed()` at every page fault outside of a paging region. As previously discussed, this implementation is inefficient and was only really intended for constant use. However, due to time constraints we could not fully improve on this.

Multicore

Our multicore booting is a multi-step process. The first step is to allocate and map the memory for all the components needed for the second core. This includes the relocatable memory segment, `core_data` segment, UMP channel and the KCB. The UMP channel is initialized to provide a simple communication channel and the current KCB is cloned to provide a baseline for the new KCB and initialize the `core_data`. Then the relocatable segment is loaded into our memory regions and all the addresses to our regions are stored in the `core_data` to let the new core know where to find all of its resources when it boots. Finally these locations are cache invalidated and cleaned to make sure the data is flushed and the new core can actually see all of our hard work! One extremely important detail is that physical addresses are required to manipulate the cache, since the cores have different address spaces. These can be obtained by using `frame_identify` on the frames. Finally, all of the regions are unmapped from our memory again except for the UMP channel, which we will need in order to use inter-core communication.

Since we have more than 1 core, the memory needs to be split so that the cores don't overwrite one another. We decided it would be sufficient to split the memory evenly among both cores. The multiboot configuration therefore contains two RAM regions and each init adds one to it's memory manager when it starts up.

To provide the other core with the capabilities to multiboot modules, the bootstrap processor (BSP) sends the frame identities of each module frame via UMP and the booting core then forges these frames based on the received frame identities.

User-level Message Passing (UMP)

Since LMP does not work between cores, we implemented User-level Message Passing to establish communication. The protocol leverages the cache coherency protocol between cores, by writing to cache line sized slots in shared memory. In our case we are using cache lines in pairs allowing us to send 63 byte sized messages.

Establishing communication between the two cores

Our initial goal was to establish message passing between the two init processes running on each core. To achieve this we allocate a shared frame (consisting of two pages) while booting the second core. The capability to this frame is then placed in a well known slot in the new init's cspace.

When booting on the second core (APP), init will map this frame and initialise a UMP channel with it. This channel is then used to receive additional information from the init running on the bootstrap processor (BSP) about the capabilities it needs to forge in order to access modules for example.

Communication using ring buffers

Every UMP channel uses two pages of shared memory, one for each direction of communication. Each page contains 64 slots consisting of 64 bytes. We were able to implement the protocol without any other shared variables.

```
struct ump_chan {
    struct frame_identity fi;
    struct ump_buf *buf;
    uint8_t buf_select;         // Buffer for this process
    uint8_t tx_counter;
    uint8_t rx_counter;
};
```

State of a UMP channel

```
struct ump_slot {
    char data[63];
    ump_msg_type_t msg_type     : 6;
    uint8_t last                : 1;
    uint8_t valid               : 1;
};
```

Struct describing a slot in the UMP ring buffer

Each participant in the UMP protocol keeps a counter pointing to the next slot to be used for sending (tx_counter) and a counter pointing to the next slot to receive on (rx_counter). These counters are not shared.

Furthermore, each slot contains a valid bit which indicates if a slot contains valid data. This data is sufficient for the protocol to work.

```c
// Receive a buffer of UMP_SLOT_DATA_BYTES bytes on the UMP channel
errval_t ump_recv_one(struct ump_chan *chan, void *buf,
                       ump_msg_type_t* msg_type, uint8_t *last) {

    // Get the correct UMP buffer
    struct ump_buf *rx_buf = chan->buf + !chan->buf_select;

    // Check if there is a new message
    if (!rx_buf->slots[chan->rx_counter].valid) {
        return LIB_ERR_NO_UMP_MSG;
    }

    // Memory barrier
    dmb();

    // Copy data from the slot
    memcpy(buf, rx_buf->slots[chan->rx_counter].data, UMP_SLOT_DATA_BYTES);
    *msg_type = rx_buf->slots[chan->rx_counter].msg_type;
    *last = rx_buf->slots[chan->rx_counter].last;

    // Memory barrier
    dmb();

    // Mark the message as invalid
    rx_buf->slots[chan->rx_counter].valid = 0;

    // Set the index of the next slot to read
    chan->rx_counter = (chan->rx_counter + 1) % UMP_NUM_SLOTS;

    return SYS_ERR_OK;

}
```

Receiving a message on a UMP channel

When attempting to receiving a message, the valid bit of the next slot is first checked, indicating if a message has been received. If the bit is set, the message can be read. However, first a memory barrier prevents the message from being read before the valid bit is checked. Another memory barrier then ensured that the message is fully read before the valid bit is cleared, indicating to the sender, that the slot can be reused. Finally the rx_counter is incremented.

```c
// Send a buffer of at most UMP_SLOT_DATA_BYTES bytes on the URPC channel
errval_t ump_send_one(struct ump_chan *chan, void *buf, size_t size,
                              ump_msg_type_t msg_type, uint8_t last) {

    // Check for invalid sizes
    if (size > UMP_SLOT_DATA_BYTES) {
        return LIB_ERR_UMP_BUFSIZE_INVALID;
    }

    // Get the correct UMP buffer
    struct ump_buf *tx_buf = chan->buf + chan->buf_select;

    // Make sure there is space in the ring buffer and wait otherwise
    while (tx_buf->slots[chan->tx_counter].valid) ;

    // Memory barrier
    dmb();

    // Copy data to the slot
    memcpy(tx_buf->slots[chan->tx_counter].data, buf, size);
    tx_buf->slots[chan->tx_counter].msg_type = msg_type;
    tx_buf->slots[chan->tx_counter].last = last;

    // Memory barrier
    dmb();

    // Mark the message as valid
    tx_buf->slots[chan->tx_counter].valid = 1;

    // Set the index of the next slot to use for sending
    chan->tx_counter = (chan->tx_counter + 1) % UMP_NUM_SLOTS;

    return SYS_ERR_OK;

}
```

Sending a message on a UMP channel

To send a message, the sender checks the `valid` bit of the next slot in the ring buffer and waits until it is cleared. When the slot becomes invalid the message is written. This is separated by a memory barrier to prevent the message from being written to a valid slot. Once the message is fully written, which is guaranteed by another memory barrier, the `valid` bit is finally set and the `tx_counter` incremented.

Clearing the `last` bit notifies the recipient that subsequent messages should be concatenated until the last bit is set. Currently this is implemented using a call to `realloc()` for every subsequent message, which is probably rather inefficient. Ideally, we would use a more sophisticated mechanism to reduce memory management costs. A possibility would be to use the `msg_type` field to encode the number of remaining messages and only send the message type in the final message.

The `msg_type` field is for use by "application layer" to differentiate between messages without the need to encode the message type in the payload, as is done with LMP. Like this we can maximise the payload size of each message and thereby the efficiency of UMP when sending large amounts of data.

Forwarding RPC calls between instances of init

Now that we have an open channel between both instances of init, we can allow RPC requests to spawn a process on another core for example.

First however, we need init to listen for both LMP and UMP messages. We achieved this by registering an additional `event_queue` on the default waitset. We then added a callback to the event queue, which then checks for UMP messages.

Now we can forward requests, which is how we handle spawning on a foreign core, process registration and URPC binding.

Spawning on a foreign core

When init receives a spawn request from a process over LMP, it parses the message and compares the requested core ID with its own. If they do not match, the it forwards the request to the init on the other core in a UMP message and waits for a UMP response.

On the other core, init receives the request, executes the spawn and returns the result over UMP. The requesting init can now pass that result back to the process that originally made the request over LMP.

This mechanism for RPC calls to foreign cores builds on the existing LMP infrastructure used to implement the `aos_rpc` library and requires both init processes to be involved. We chose this approach for reasons of simplicity. It could be significantly optimised by allowing processes to bind directly to either init. However, currently our process management does not explicitly track the init processes, meaning that both instances of init carry the implicit PID of 0. This is also the reason why our URPC API does not support binding to any instance of init.

The ideal solution would be to move all of init's functionality (spawning, memory management, etc.) to separate processes and then use our URPC API to bind to each of these, removing the need for duplication of some of these services across cores.

Contrast to LMP

Unlike LMP, this UMP does not currently support zero-copy sending. However, this could be implemented by sending frame identities and forging the corresponding frame capabilities. On the other hand, this would always require the involvement of init, which would make it impractical for use with URPC (discussed later).

While UMP is considerably faster than LMP when it is used between cores, it is almost unusably slow when used on the same core, due to the absence of involvement with the scheduler. This is what lead us to implement the URPC API to unify both protocols seamlessly.

User-level Remote Procedure Calls (URPC)

In light of the performance considerations of LMP and UMP discussed above and the need for message passing between processes beyond init, we decided to create a unified API which could allow arbitrary forms of RPC calls between arbitrary processes, regardless of the "transport" protocol (LMP or UMP).

With this goal in mind, we designed the following API:

```
// Bind to a URPC server with a specific PID
errval_t urpc_bind(domainid_t pid, struct urpc_chan *chan, bool use_lmp);

// Accept a bind request if one was received and set up the URPC channel
errval_t urpc_accept(struct urpc_chan *chan);

// Accept a bind request and set up the URPC channel
errval_t urpc_accept_blocking(struct urpc_chan *chan);

// Send on a URPC channel
errval_t urpc_send(struct urpc_chan *chan, void *buf, size_t size,
                   urpc_msg_type_t msg_type);

// Receive on a URPC channel
errval_t urpc_recv(struct urpc_chan *chan, void **buf, size_t *size,
                   urpc_msg_type_t* msg_type);

// Blockingly receive on a URPC channel
errval_t urpc_recv_blocking(struct urpc_chan *chan, void **buf, size_t *size,
                            urpc_msg_type_t* msg_type);
```

URPC API

Using `urpc_bind()` a process (the *client*) can open a new URPC channel by specifying the recipient process's PID and if LMP or UMP should be used. This is the only call, where this distinction needs to be made by a client of API. Automatic selection of the transport protocol could be implemented, but would require an additional request to init or a somewhat more complicated implementation of the binding mechanism. In our current usages this was never necessary and if it was one could always default to UMP (accepting considerable loss in performance when the processes are running the same core).

`urpc_accept()` and `urpc_accept_blocking()` allows a process (the *server*) to accept an incoming binding request. The transport protocol is specified in the request and set accordingly.

Once a URPC channel is established between two processes they can freely invoke `urpc_send()` to send and `urpc_recv()` or `urpc_recv_blocking()` to receive messages of arbitrary length. The message_type can be used to differentiate between messages.

Binding over UMP

When attempting to bind with a process over UMP, the client allocates a new UMP frame and sends the frame capability to init over LMP together with the server's PID. Next, the client waits on an ACK message from the server using `urpc_recv_blocking()`, concluding the binding process.

After receiving the LMP message, init looks up the process information and checks which core the server is running on. If it is the same core, init simply forwards the message to the server. Otherwise it retrieves the frame identity and forwards this information and the PID to the other instance of init, which then forges the frame capability and forwards it to the server process.

Once the server receives the frame capability it maps the frame, initialises the UMP channel and sends the acknowledgement using `urpc_send()`.

Binding over LMP

The process of binding over LMP is slightly more involved than over UMP, but follows the same concept and leverages the identical code in init.

First, the client creates a new LMP endpoint using `lmp_chan_accept()` and sends the endpoint capability to init in place of the frame capability which is sent when using UMP.

Again, init verifies the PID received in the request and drops the request if it is destined for another core, as LMP only works on the same core. Otherwise init forwards the request and attaches the source PID to the forwarded request.

Upon receiving the request, the server also creates an LMP endpoint by calling `lmp_chan_accept()`. Using the PID of the client, it sends the endpoint back, allowing the client to finish the initialisation of it's LMP channel.

After the client's LMP channel is initialised it sends an acknowledgment to the server over the LMP channel. The server then finishes the binding process by sending the same acknowledgment as with UMP by means of `urpc_send()`, ensuring that the URPC channel is fully set up.

PID discovery

The decision to use PIDs for binding was mostly based on the need for a unique identifier for processes, given the lack of a name-server. However, depending on the use-case, the PID of a target process may not be well know. To this end, we implemented a simple method for PID discovery by name: `aos_rpc_process_get_pid_by_name(const char *name, domainid_t *pid)`. This method is particularly useful when binding to services. It uses an RPC call to get an array of all PIDs of running processes. It then iterates the array and makes an RPC call for each PID to get the name of the process.

Sending and receiving

When using UMP, messages are sent and received as described in the UMP section above. However, when using LMP, the process is much more complex.

Depending on the length of the buffer to be sent, the buffer is either encoded into the arguments of an LMP message or sent using a shared frame, meaning that in either case only one message is necessary. The implementation automatically switches between these functionalities by calling `lmp_send_buffer()` for sending and `lmp_recv_buffer()` for receiving.

This family of functions is very similar to the `lmp_send_string()` and `lmp_recv_string()` functions described in the LMP section, except that they use optimised implementations of the underlying transport functions (`lmp_send_short_buf_fast()`, `lmp_send_frame_fast()`, etc.), which do not wait for acknowledgments from the receiver. This could cause sends to fail if the receiver does not poll for messages quickly enough. However, the performance loss makes URPC over LMP slower than URPC over UMP, which is unacceptable.

Conclusion

Overall, the URPC API has proven to be extremely useful and was used in all of our individual projects. Most notably, it provides a direct abstraction for sockets in the network stack. It is also significantly simpler to use than just bare UMP or LMP channels.

On the other hand, the API does not support sending capabilities in any way. This could be tricky to add, since it isn't supported in UMP and some involvement of the init process would be necessary. Due to how process management is currently structured, the API also does not support binding with any instance init. However, if these features were implemented, the API would provide the ideal foundation to implement functionality such as memory management, which currently uses the `aos_rpc` calls.

TurtleSHELL (Sven Knobloch)

// Something introductory

The shell implementation (nicknamed "TurtleSHELL")

State

```
struct io_buffer {
    char *buf;
    size_t pos;
    struct io_buffer *next;
};
```

IO Buffer

```
struct terminal_event {
    urpc_msg_type_t type;
    struct urpc_chan *chan;
    struct terminal_msg msg;
};
```

Terminal Event

```
struct terminal_state {
    void *write_lock;
    void *read_lock;
    struct io_buffer *buffer;
    collections_listnode *urpc_chan_list;
    collections_listnode *terminal_event_queue;
};
```

Terminal State

The core of the terminal driver is stored in it's state object. The `read_lock` and `write_lock` pointers are pointers to the respective channel that currently holds the lock on the terminal. Since the terminal is single-threaded there is no need for a complicated lock and therefore provides a simple solution to interleaved writing and reading. The `io_buffer` is a linked list of buffers that holds the buffered input characters read via the serial interrupts. The `urpc_chan_list` contains a linked list of URPC channels that have registered with the terminal driver. Finally, the `terminal_event_queue` holds a linked list of `terminal_events` which can be traversed to either handle or defer the events. This state is initialized at the start of the terminal process and passed around to the list predicates and the interrupt handler to maintain a consistent state without relying on statics or globals.

Userspace Serial Driver

```
// Get and map Serial device cap;
lvaddr_t vaddr;
err = map_device_register(OMAP44XX_MAP_L4_PER_UART3, OMAP44XX_MAP_L4_PER_UART3_SIZ
if (err_is_fail(err)) {
    debug_printf(err_getstring(err));
}

mem = (char *) vaddr;
flag = (char *) vaddr + 0x14;
```

Serial Address Mapping

```
void put_char(char c) {
    while (!(*flag & 0x20));
    *mem = c;
}

static char get_char(void) {
    while (!(*flag & 0x1));
    return *mem;
}
```

IO Functions

The serial driver uses device capabilities to run in the userspace. It starts by requesting these capabilities from the kernel via the `map_device_register` helper function which also maps the addresses into the virtual address space. The two IO functions, `get_char` and `put_char`. These functions are extremely similar to the ones from the Milestone 0 assignment, but provide much better performance as the interrupt eliminates the need to constantly spin on the flag.

Interrupt Handler

```
static void serial_interrupt_handler(void *arg) {
    struct terminal_state *state = (struct terminal_state *) arg;

    char c = get_char();

    // Iterate to tail
    struct io_buffer *current;
    for (current = state->buffer; current->next != NULL; current = current->next)
```

```
        // Set char and inc
        current->buf[current->pos++] = c;

        // Alloc new list if full
        if (current->pos == IO_BUFFER_SIZE) {
            io_buffer_init(&current->next);
        }
    }
```

The interrupt handler is set directly after the `terminal_state` and `io_buffer` are initialized. The state's `io_buffer` stores the sequence of characters entered into the picocom terminal and is structured as a linked list so that additional buffer space can easily be appended. This allows for virtually infinite buffering. The interrupt handler then has an extremely easy job of just inserting the new characters as they become available. Only downside is that it runs on the same thread as the terminal polling and therefore has occasional latency.

Terminal Driver

```
static int terminal_event_dispatch(void *data, void *arg) {

    // Trivial assignments, casts and error handling are removed

    switch (event->type) {
        case URPC_MessageType_TerminalReadLock:
            if (st->read_lock == NULL || st->read_lock == event->chan) {
                st->read_lock = event->chan;
                err = urpc_send(event->chan, (void *) &msg,
                        sizeof(struct terminal_msg), event->type);
                return 1;
            }
            break;
        case URPC_MessageType_TerminalRead:
            if (st->read_lock == NULL || st->read_lock == event->chan) {
                msg.err = get_next_char(st, &msg.c);
                if (msg.err == SYS_ERR_OK) {
                    err = urpc_send(event->chan, (void *) &msg,
                            sizeof(struct terminal_msg), event->type);
                    return 1;
                }
            }
            break;
        case URPC_MessageType_TerminalReadUnlock:
            if (st->read_lock == NULL || st->read_lock == event->chan) {
                st->read_lock = NULL;
            }
            err = urpc_send(event->chan, (void *) &msg,
                    sizeof(struct terminal_msg), event->type);
            return 1;
        case URPC_MessageType_TerminalWriteLock:
            if (st->write_lock == NULL || st->write_lock == event->chan) {
                st->write_lock = event->chan;
                err = urpc_send(event->chan, (void *) &msg,
                        sizeof(struct terminal_msg), event->type);
                return 1;
            }
            break;
        case URPC_MessageType_TerminalWrite:
            if (st->write_lock == NULL || st->write_lock == event->chan) {
                put_char(event->msg.c);
                err = urpc_send(event->chan, (void *) &msg,
                        sizeof(struct terminal_msg), event->type);
                return 1;
            }
            break;
        case URPC_MessageType_TerminalWriteUnlock:
            if (st->write_lock == NULL || st->write_lock == event->chan) {
                st->write_lock = NULL;
            }
            err = urpc_send(event->chan, (void *) &msg,
                    sizeof(struct terminal_msg), event->type);
            return 1;
        case URPC_MessageType_TerminalDeregister:
            collections_list_remove_if(st->urpc_chan_list,
                    urpc_chan_list_remove, event->chan);
            err = urpc_send(event->chan, (void *) &msg,
                    sizeof(struct terminal_msg), event->type);
            return 1;
            break;
    }

    return 0;
}
```

Terminal Event Handling

```
struct terminal_event {
    urpc_msg_type_t type;
    struct urpc_chan *chan;
    struct terminal_msg msg;
};
```

Terminal Event

The terminal driver is set up as a new process to prevent it from bogging down `init`. It acts as a server that clients can bind to with several different functionalities. These functionalities are provided as a URPC calls that push the request into an event queue. The events encapsulate the message contents, type and sender. The queue is then polled using a predicate function that will try to resolve the event and remove itself from the queue. The queue implementation enables the processes to block on the call instead of spinning on the URPC calls, which is a large performance boost.

Registration

In order to use the terminal server, any process must first bind with it over URPC. This is done in the `aos_rpc_init` call (except for terminal itself since a process can't bind to itself) and the `urpc_chan` to the terminal is then saved in the `aos_rpc_chan` state. Once registered the terminal driver will continuously poll the channel for new messages.

Locking/Unlocking

When using the terminal server, processes have the ability to lock/unlock the terminal to prevent output from interleaving. The processes must make the URPC call and then the terminal prevents all other channels from reading/writing until the owning process unlocks it with the corresponding URPC call.

Read/Write

Reading and writing is the core feature and allows the processes to communicate. Processes can read and write single characters as long as they own or no one owns the lock.

Deregistration

In order to prevent the terminal server from spinning on dead processes, processes will deregister themselves upon executing the `exit` function. They are then removed from the list of active channels and will no longer be able to use the terminal.

Shell

The shell implementation is a bit rudimentary. It features a somewhat sophisticated argument parsing as well as a select list of implemented commands and tracking of the current directory. The shell also will print all input it receives back to the console.

The argument parsing is done with two buffers. The raw input is read into the first buffer and is then parsed into the second. The parsing supports escaped characters and both single and double quotes.

Also, some more rpc calls were added to facilitate the command and general shell functionality. The `aos_rpc_process_deregister_notify` allows processes to listen for another process to exit while blocking. This lets the shell provide a more sequential interface but processes can still be dispatched to the background using the & symbol.

Commands

The following commands are supported by the system, along with a short description of what they do:

- help - Prints a help menu for all available commands
- echo [string] - Prints string
- ls (path) - Lists all files in the given directory or the current directory
- cat [filename] - Prints contents of file
- mkdir [path] - Makes directory at path
- rmdir [path] - Deletes directory at path

- touch [filename] - Makes file at path
- rm [filename] - Deletes file at path
- ps - Prints list of all processes
- time [cmd] (args...) - Measure the time in ns it takes to execute a command
- exit - Exit the shell
- [elf name] (args...) - Run a program with the given name and arguments

Filesystem (Sebastian Winberg)

Introduction

The implementation of the FAT filesystem is based on a service structure. User-level processes initially bind with the mmchs service using URPC and send RPC request to the service. The service thereby constantly non-blockingly accepts requests, processes them and then sends the some information about the file back to the client. Due to this structure write atomicity, can be guaranteed, since the service is only working on one directory or file change at a time.

Device capability

In order for the SD block driver to work we had to implement the following AOS RPC call:

```
errval_t aos_rpc_get_device_cap(struct aos_rpc *chan, lpaddr_t paddr, size_t byte
                                struct capref *frame);
```

This call is supposed to give back a device frame that covers a special physical address region where certain device registers are located. To interact with the device registers, one must only map the devframe into virtual address space. It thereby has to be mapped *non-cachable* since we are not dealing with memory, but with device registers that could change their values between two consecutive reads.

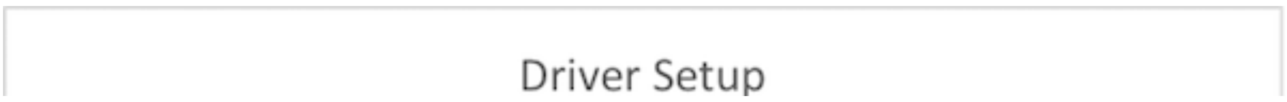
We handle this AOS RPC call using `LMP_RequestType_DeviceCap`. Init thereby receives a range for which the user process requests a device capability. Init then gets the device capability through `cap_devices` (with slot `TASKCN_SLOT_DEVCAP`) that was initialized on startup and then invokes the frame_identity to get the devframe capability base and bytes (size).

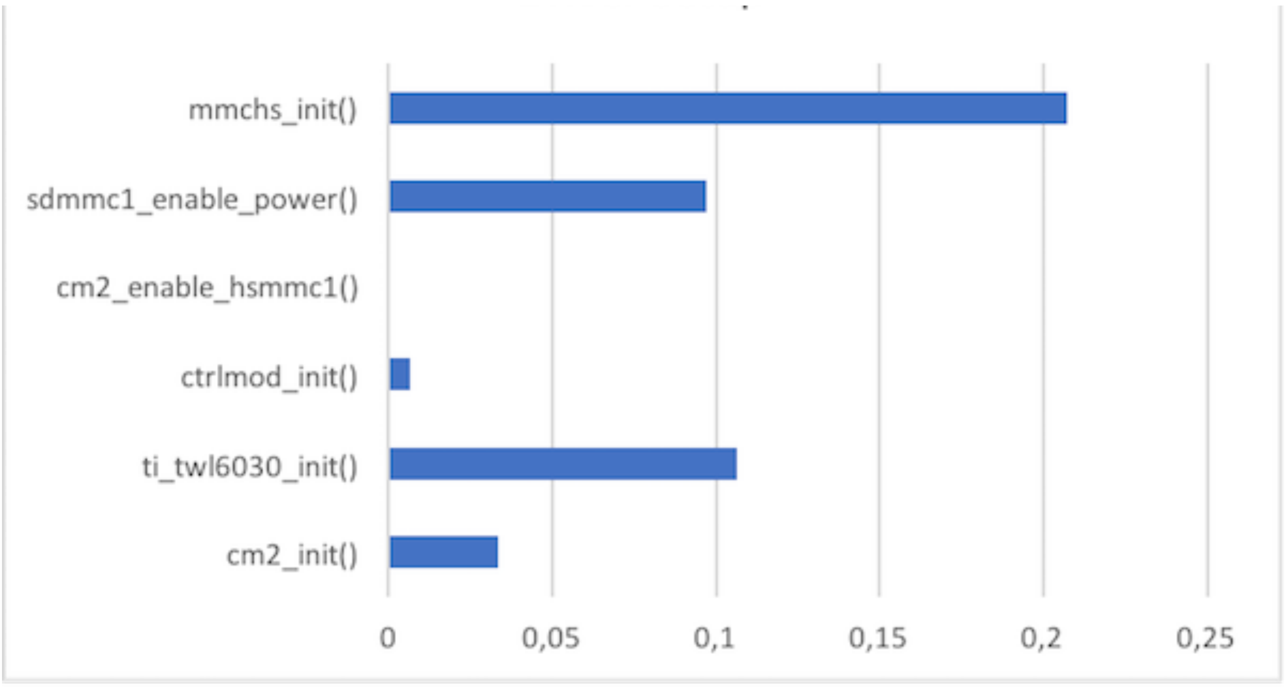
After some checking that the requested paddr and bytes are sufficient and lie in the overall device frame range, the device capability is retypes with the requested range of [paddr, paddr + bytes - 1] to the capability type `ObjType_DevFrame`. The LMP server then send back this newly retyped capability to the client and afterwards deletes the capability and frees the slot.

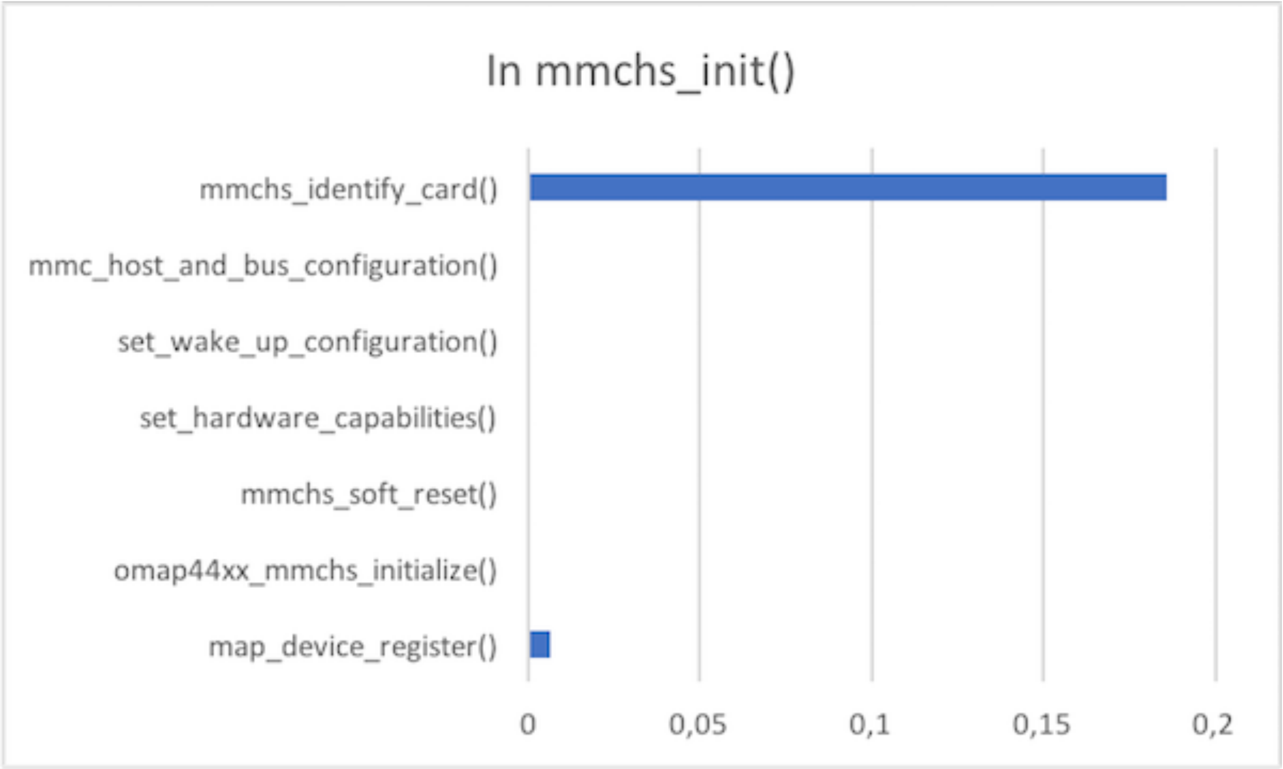
With this AOS RPC call implemented the mmchs block device driver for the SD card works, unless you forget to put a SD card into your PandaBoard, in which case a non-descriptive assertion is triggered.

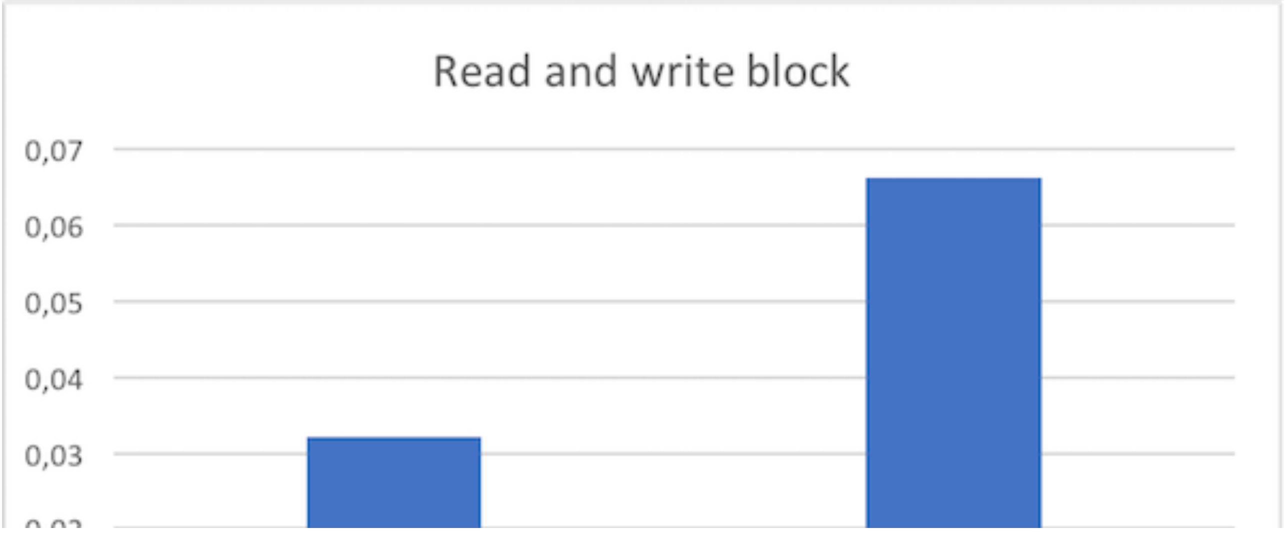
Block Driver Benchmark

For the SD card block device driver *mmchs* we have the following benchmarking results:




mmchs_init()
sdmmc1_enable_power()
cm2_enable_hsmmc1()
ctrlmod_init()
ti_twl6030_init()
cm2_init()
0
0,05
0,1
0,15
0,2
0,25


In mmchs_init()
mmchs_identify_card()
mmc_host_and_bus_configuration()
set_wake_up_configuration()
set_hardware_capabilities()
mmchs_soft_reset()
omap44xx_mmchs_initialize()
map_device_register()
0
0,05
0,1
0,15
0,2


Read and write block
0,07
0,06
0,05
0,04
0,03

0,02
0,01
0
mmchs_read_block()
mmchs_write_block()

the time has been obtained using the aos/systime.h library

Read and write performance of the mmchs block functions is rather inefficient. This is due to polling (waiting) for the block to be read / written. `mmchs_write_block()` thereby takes approximately twice as long with 66.27ms as the `mmchs_read_block()` with 32.18ms. Furthermore is the blocksize of 512 bytes an additional delimiting factor.

As one can see in the graphs is the mmchs driver mostly busy identifying the card. As noted by some students does SD card driver have quite a lot of issues in general. While working on the project I used my own 16 GB SD card, that I formatted the following way:

```
$ mkfs.vfat -I -F 32 -S 512 -s 8 /dev/sdX
```

I did not have any issues with the SD card driver, but when my fellow team mates tried to use the SD cards given by the AOS assistance, the driver returned zeroed out blocks, even though the SD card was formatted the same way.

Library Code Structure

The code for the filesystem implementation is structured in two libraries:

- A client side library `lib/fs` responsible for:
  - mounting of ramfs, fatfs and mbtfs
  - initial service binding and client communication to mmchs
  - virtual file system multiplexing (VFS)
- A service side library `lib/fs_serv` responsible for:
  - mmchs FAT filesystem service
  - non-blocking accepting of different RPC requests using URPC
  - actual read and writing from FAT blocks / sectors and clusters

Due to the size of the libraries we present an overall structure of the files used in these two libraries.

The following files thereby contain different parts of the libraries:

Files for client	Description	Example functions
`vfs.c`	Virtual file system layer	`vfs_open()`
`ramfs.c`	Ram file system	`ramfs_open()`
`mbtfs.c`	Bootinfo file system	`bfsfs_open()`
`fs_rpc.c`	FAT file system (RPC calls)	`fs_rpc_open()`
`fopen.c`	File functions (fs_libc)	`fs_libc()`
`dirent.c`	Directory functions	`diropen()`
`fs.c`	Filesystem init and mounting	`filesystem_init()`

Files for service	Description

`fatfs_rpc_serv.c`	FAT file system service
`fatfs_serv.c`	FAT cluster read/writing
`fat_helper.c`	Name convertion functions

FAT filesystem service

The FAT filesystem service has to be spawned in the shell using the following command:

```
DoritOS $USER: oncore 0 mmchs &
```

In order for the shell to be able to use the filesystem we implemented the call

```
DoritOS $USER: fs_init
```

to mount the SD card in the directory `/sdcard`, the multiboot modules in `/multiboot` and ramfs in the root directory `/`. The actual command thereby calls the `filesystem_init()` in `fs/fs.c` which mounts the different filesystems and sets up the VFS state. Note that each process has to mount it for themselves. The gool with the filesystem service was to make it as stateless as possible. Therefore all mounting and file descriptor handling is done by the process in the `fs` library.

The RPC calls for both service and client are send over a buffer using URPC Thereby any arguments passed in a wrapper `struct fs_message` which is stored in the beginning of the buffer. Any additional information, like a string path or directory entry `struct dirent` is passed right after the four arguments in the same buffer.

Virtual File System Layer

The filesystem implementation has a VFS layer to support multiple different file system implementations with different mounting directories. The VFS abstraction layer thereby is between `fopen.c` with the `fs_libc` functions, that handle the file descriptor table and the file descriptors respectively, and the filesystem specific implementation of these functions, being for example `ramfs_open()` (for ramfs) or `fs_rpc_open()` (for fatfs).

The VFS handle thereby stores a void pointer to the filesystem specific handle together with the filesystem specific type for multiplexing.

We thereby implement the special VFS state structure:

```
struct vfs_mount {
    void *ram_mount;
    void *fat_mount;
    void *mbt_mount;
    struct mount_node *head;
};
```

It contains pointers to all specific filesystem states that will be passed on to the filesystem specific implementation of the respective function. Furthermore does it include a header to the mounting structure. The mounting datastructure is in our case a in decreasing lexicographically order sorted linked list of all mounted directories. Due to the fact that it is sorted this particular way, we can easily compare the node name with the path prefix.

If we look at the example mounting linked list:

```
[t1, “/sdcard/folder“] -> [t2, “/sdcard”] -> [t3, “/”]
```

One can see that iterating the list and taking the *first* node with a fully matching name is a sufficient implementation due to the ordering property. The search throught the list will be done in $O(n)$, with n being the number of mounted directories. Given the fact that most used cases have only a small constant amount of mounted directories, this is a reasonable asymptotic complexity.

For each `open`/`opendir` the VFS layer looks up which mount node name has the longest matching prefix with the given path. The specific implementation of the longest matching prefix algorithm can be seen in the following code snippet:

```c
enum fs_type find_mount_type(struct mount_node *head, const char *path,
                             char **ret_path) {
    assert(ret_path != NULL);

    // Set temp to head of mount linked list
    struct mount_node *temp = head;

    // Set default type to RAMFS
    enum fs_type ret_type = RAMFS;

    while(temp != NULL) {

        // Check if prefix of path matches temp's name
        if (strncmp(path, temp->name, temp->len) == 0) {

            // Set return mount type
            ret_type = temp->type;

            // Set return path to suffix
            *ret_path = strdup(path + temp->len);

            return ret_type;
        }
        // Update temp
        temp = temp->next;
    }

    // Set return path to duplicate of path
    *ret_path = strdup(path);

    return ret_type;
}
```

Mounting longest matching prefix function

The function thereby returns the path relative to the mounted directory, which is passed on to the filesystem specific function together with the filesystem specific state in `vfs_mount`.

Bootinfo mounting

After the VFS layer was properly structured we were now able to properly multiplex `fs_libc_open` with either `ramfs_open` or `fs_rpc_open`. To further widen the possibility for this abstraction layer I implemented the read only file functions for the multiboot modules. In mbtfs.c we have fairly straight forward implementation of the basic file functions like `fopen` or `fread`.

On the filesystem initialization it thereby requests a list of all module names from init using the newly created request:

```
errval_t aos_rpc_get_module_list(struct aos_rpc *chan,
                                 char ***modules,
                                 size_t *module_count);
```

This request returns an array of module names and a count of all modules, which are then saved in the filesystem state `struct mbtfs_mount` for later reference. Behind the scene this call requests the module name list inside a buffer using using LMP to init. Keep in mind that both init's (on core 0 and 1) are aware of all multiboot modules since they are passed in the bootinfo structure, which allows us to ask the init on our core.

If we now want to open a multiboot filesystem file another AOS RPC call is called:

```
errval_t aos_rpc_get_module_frame(struct aos_rpc *chan, char *name,
                                  struct capref *frame, size_t *ret_bytes);
```

This call requests the frame capability for the module with given name from init using LMP. On receiving on the server side init looks up the mem_region using `multiboot_module_name()`. It then creates a capref out of that and sends it back to the client using LMP.

The client side can then map the received module frame to virtual address space and saves the frame and buffer alongside some other information like the current position and size in the `struct mbtfs_handle`.

ELF Loading

Due to time constraints and design decisions we have made earlier on, that caused problem arising with URPC, I was unfortunately not able to implement ELF loading of files from the SD card.

The problem arised with sending over the data to the spawn service. In an earlier milestone we implemented the spawn service as part of the init process. This came to our disadvantage when approaching the ELF loading, since you are unable to bind with init using the URPC message passing.

Therefore the mmchs filesystem service is not able to directly send over the data in order for init to load and spawn the ELF, unless we differentiate between LMP and UMP.

Furthermore did the problem arise that copying over big chunks of data like an ELF file is a rather costly and time consuming operation.

Because of this we also thought about passing init a frame identity, with which it would then be able to forge, map the file, load the ELF and then spawn the process. Unfortunately we did not have enought time to implement this.

Benchmark of filesystem

We tried to use the omap timer given in the filereaderto make benchmarkings for the filesystem, but every time we used it, the entire OS got stuck. As we found out one day before the deadline, this is due to a more recent change in barrelfish, where the scheduler uses this timer. Setting initializing it will therefore stop the scheduler from working.

Due to this rather late discovery of the bug and no knowledge of the aos/systime.h library prior to that, I was unfortunately not able to do a benchmark on my filesystem.

Some more general marks regarding the performance are the following: Since my filesystem implementation is always reading from the SD card and writing through the changes back, the system is rather slow. Furthermore is a lot of memory allocation and copying done which further slows down the process.

Future improvements would be to store the used region of the file allocation table in main memory. Emphasis is thereby put on used, since loading the entire FAT table into main memory took more than 10 minutes, when I tried it. The `FSI_Nxt_Free` value in the `FAT FSInfo Sector Structure` is thereby a good indicator, on how much of the file allocation table *actually* has to saved in main memory.

Conclusion

Writing a working filesystem implementation is a lot of work and involves quite a lot of indepth knowledge about the internals of both the operating system you are implementing it in, and the file system specifications (in this case FAT32). Especially the walking of the file allocation table and the consideration of all corner cases makes part of the code very extensive.

Overall was this project one of the most challenging tasks I have worked on.

Network Stack (Carl Friess)

The network stack has a very modular structure. The implementations for SLIP, IP, ICMP and UDP are all separated and (as far as is reasonable) only use methods from the next network layer. Client processes access the network using UDP sockets and the `net` library. UDP sockets consist mostly of a URPC channel and a port.

All interactions with the network run through the `networkd` service, which is automatically executed at startup. Most of `networkd`'s code is designed to be as portable as possible.

Serial - Physical Layer

The physical layer is provided by an additional serial connection (UART4). The driver for this was provided but required access to the device frame for the serial controller. The device frame is retrieved using the `aos_rpc_get_device_cap()` RPC call and mapped uncachable. For it's initialisation to succeed, it also requires the IRQ capability, which is normally only available in init. For simplicity, the capability is just copied into every process's cspace.

The serial controller has a 64 byte input FIFO. An interrupt is triggered when one byte is inserted into the FIFO. The interrupt event is dispatched on the main waitset and reads as much data as is presentin the FIFO, before directly passing it to the SLIP parser.

Overall this mechanism worked well. However, the interrupts were seemingly not delivered fast enough, so that it was necessary to decrease the BAUD rate of the serial connection to 9600 in order to be able to read incoming data faster than it was delivered and thereby avoid a buffer overrun.

Serial Line Internet Protocol (SLIP) - Link Layer

The incoming data is parsed as it comes in and stored in a fixed size buffer. The parser unescapes any escape sequences. When a `SLIP_END` byte is received, it invokes a handler on the IP layer to process the received packet. Empty packets are immediately discarded.

The `slip` module also provides the `slip_send(uint8_t *buf, size_t len, bool end)` method, which escapes and sends a buffer, optionally terminating the packet with a `SLIP_END` byte.

Internet Protocol (IP) - Internet Layer

When invoking the incoming packet handler, the packet is first checked for a valid length. Next, the IP header is parsed and all fields are stored in a struct and where necessary, converted from network order to little-endian. It's probably unnecessary to parse the full header, but it provides more clarity in subsequent code. Once the header is parsed, the checksum is computed (using the `netutil/checksum` library) and if faulty, the package is dropped.

Now the packet can be passed on to the next module based on the protocol number in the IP header.

Outgoing packets are sent from the configured IP address, which can be set using the `ip_set_ip_address(uint8_t a, uint8_t b, uint8_t c, uint8_t d)` method.

Internet Control Message Protocol (ICMP) - Transport Layer

Again, the ICMP header is parsed similarly as on IP layer and the checksum is computed. Then the message type is evaluated. Currently only type 8 (echo request) is supported. All other messages are dropped.

Echo requests are handled by sending back the same message but with type 0 (echo reply). First, an IP header is constructed and sent by calling the `ip_send_header(uint32_t dest_ip, uint8_t protocol, size_t total_len)` method of the `ip` module. Next a new `struct icmp_header` is populated and the checksum for the entire message computed. Finally, the header is encoded and sent with the ICMP payload using `ip_send(uint8_t *buf, size_t len, bool end)`.

User Datagram Protocol (UDP) - Transport Layer

When handling an incoming datagram, the UDP header is first parsed in a similar manner as in the `ip` and `icmp` modules, but the computation of the checksum is slightly more involved, as a pseudo IP header needs to be constructed and included in the checksum. I solved this by adding the following struct which accurately lays out the pseudo header and can be used to complete the checksum computation.

```
struct udp_checksum_ip_pseudo_header {
    uint32_t src_addr;
    uint32_t dest_addr;
    uint8_t zeros;
    uint8_t protocol;
    uint16_t udp_length;
    uint16_t checksum;  // Checksum of UDP header and payload
} __attribute__ ((__packed__));
```

Pseudo IP header for UDP checksum computation

Once the packet has been validated, it is passed on to a client process or dropped. This multiplexing of UDP packets is achieved using the socket model.

UDP sockets

I implemented the `net` library to provide an API for interaction with `networkd` roughly based on the UNIX socket API.

```
// Open a UDP socket and optionally bind to a specific port
//  Specify port 0 to bind on random port.
errval_t socket(struct udp_socket *socket, uint16_t port);

// Blockingly receive a UDP packet on the given socket
errval_t recvfrom(struct udp_socket *socket, void *buf, size_t len,
                  size_t *ret_len, uint32_t *from_addr, uint16_t *from_port);

// Send a UDP packet on the given socket to a specific destination
errval_t sendto(struct udp_socket *socket, void *buf, size_t len,
                uint32_t to_addr, uint16_t to_port);

// Close a UDP socket
errval_t close(struct udp_socket *socket);
```

Socket API provided by the net library

The socket API is built on top of the URPC API. Each socket instance is essentially just a wrapper around a URPC channel, as can be seen below. Thanks to URPC's binding mechanism, it's possible to easily set up multiple URPC channels between two processes, meaning that a client of the `net` library can hold multiple sockets without complicated multiplexing on either the client side or in `networkd`.

```
struct udp_socket_common {
    uint16_t port;
};
struct udp_socket {
    struct udp_socket_common pub;
    struct urpc_chan chan;
};
```

Data structure describing a socket (client side)

```
enum udp_socket_state {
    UDP_SOCKET_STATE_CLOSED,
    UDP_SOCKET_STATE_OPEN
};
struct udp_socket {
    struct udp_socket_common pub;
    struct urpc_chan chan;
    enum udp_socket_state state;
};
```

Data structure describing a socket (networkd)

Opening sockets

In order to send or receive the client must always first open a socket using the `socket()` call. The client can bind to a specific port or have one allocated by `networkd`. Opening a socket will first make a bind request to `networkd` and then send a `URPC_MessageType_SocketOpen` message over the newly created URPC channel.

`networkd` holds a linked list of open socket. These are initially in the `UDP_SOCKET_STATE_CLOSED` state (after binding has completed). On receiving a `URPC_MessageType_SocketOpen` message, `networkd` will check the list for any other sockets holding the requested port or just allocate an unused port. If successful it will change the state of the socket to `UDP_SOCKET_STATE_OPEN` allowing packets to be forwarded on this socket's URPC channel.

Receiving datagrams

Whenever a UDP datagram is received, `networkd` searches the linked list for a socket bound to the destination port. If one is found, the datagram is forwarded over the socket's URPC channel. Otherwise, the packet is dropped.

Rather than forwarding the entire packet, the source address, source port and payload are forwarded using the structure below. This provides the client with all the necessary information to send back a response.

```
struct udp_urpc_packet {
    uint32_t addr;
    uint16_t port;
    uint16_t size;
    uint8_t payload[];
} __attribute__ ((__packed__));
```

struct for forwarding a datagram over URPC

Clients can use `recvfrom()` to receive incoming UDP packets. `recvfrom()` simply uses `urpc_recv_blocking()` to wait for a `URPC_MessageType_Receive` message on the socket's URPC channel and returns the data and payload contained in the `struct udp_urpc_packet`.

Sending datagrams

To send a datagram clients can use the `sendto()` method, which encodes the packet in the same `struct udp_urpc_packet` format and sends it in a `URPC_MessageType_Send`. Now the `addr` and `port` fields describe the destination rather than the source, which was the case for incoming packets.

To handle these incoming messages (as well as bind requests and other messages), `networkd` needs a way to check for new messages on all the URPC channels attached to sockets. I solved this by adding an `event_queue` to the default waitset which contains a handler. It first checks for a new bind request and then iterates the list of sockets and calls `urpc_recv()` on each socket's URPC channel.

This could have been solved by using a simple while loop but the events on the default waitset must be dispatched in order for the serial interrupts to be delivered. This could have been done by calling `event_dispatch_non_block()` but I found the solution described above to be more elegant.

Once `networkd` has received a `URPC_MessageType_Send` message it sends an IP header, composes and encodes the UDP header, computes the UDP checksum and finally sends the UDP header and payload. The source port is set to the port the socket is bound to.

Closing sockets

Closing a socket is a simple operation but very important so that ports are not blocked until the system restarts. The `close()` method simply sends a `URPC_MessageType_SocketClose` message to `networkd` causing the socket to be deleted from the linked list. Unfortunately the URPC API currently does not provide a way to clean up dead URPC channels due to the lack of underlying support for cleaning up UMP and LMP channels. This means that closing sockets will leak some resources.

UDP echo server

Using this API I implemented a simple UDP echo server (`udp_echo`) which makes use of all of the API's features. It can be launched without an argument or with a port as first argument.

Performance issues

As previously described, the serial port provided some performance limitations, but the UDP event handler on the default waitset did not improve things. In fact, for messages greater than 64 bytes (where performance issues start to occur due to the serial FIFO's size) it became necessary to cancel the UDP handler on the event queue until the incoming packet is received.

To implement this I added an "idle" state to the SLIP parser. When exiting this state, the parser calls `udp_cancel_event_queue()`. After re-entering the idle state, the parser calls `udp_register_event_queue()` to re-enable UDP event handling. This works almost all of the time but is really just a work-around and has only been properly tested at 9600 BAUD. The issue is, that the parser could (with unluckily timing) miss the beginning of a package, at which point it's already too late. A better solution would be to use Request To Send and Clear To Send flow control signals on the serial line or even DMA rather than just the RX FIFO.

Overall, the performance is reasonably good though and given packets smaller than 64 bytes the network stack works flawlessly.

Network configuration

networkd can be configured dynamically at runtime. Currently the static IP address can be set and dumping of raw packages can be enabled using the following network utilities.

ip_set_addr

This simple application opens a socket to establish communication with networkd and then sends a special URPC_MessageType_SetIPAddress message, which contains the four bytes making up the new IP address. On receiving this message, the ip_set_ip_address() method is called and all future packets are sent from the new IP address.

dump_packets

Similar to the previous utility this application sends a URPC_MessageType_DumpPackets containing just a boolean value. It tells the SLIP parser whether to dump incoming packets to the stdout after they have been fully parsed.

udp_send

This is a very simple program which opens a socket, sends the message passed as an argument to the address and port specified in the arguments, closes the socket and terminates.

Remote Shell

We implemented a basic version of a remote shell by essentially redirecting the stdin and stdout of processes over UDP packets to an instance of netcat on the remote host.

Redirecting stdin and stdout

Rather than implement a system similar to the TTY subsystem in Unix, we decided to leverage the versatility of our URPC API. Our idea was to simply have multiple processes implement the functionality of the terminal server. Processes then bind to the appropriate process depending on how their stdin and stdout should be routed.

We started by adding a terminal_pid field to the Dispatcher Control Block (DCB) and introducing an aos_rpc spawn call which allows the PID of the terminal server for the new process to be specified.

```
/**
 * \brief Request process manager to start a new process
 * \arg name the name of the process that needs to be spawned (without a
 *           path prefix)
 * \arg terminal_pid the process id of the terminal process to be used by the
                    new process
 * \arg newpid the process id of the newly spawned process
 */
errval_t aos_rpc_process_spawn_with_terminal(struct aos_rpc *chan,
                                             char *name,
                                             coreid_t core,
                                             domainid_t terminal_pid,
                                             domainid_t *newpid);
```

AOS RPC spawn call with specific terminal PID

When set to 0, init will choose the PID of the default terminal server. However in the case of the shell, it can simply pass on its own terminal's PID. This way a newly spawned process will "inherit" the shell's terminal.

```
// Spawn process
domainid_t pid;
err = aos_rpc_process_spawn_with_terminal(aos_rpc_get_init_channel(), buf, 1,
                                          disp_get_terminal_pid(), &pid);
```

Terminal PID inheritance as done in the shell

Adding another terminal server

To simplify achieving multiple terminal servers, we decided to move the core terminal functionality into a library (`libterm`). The implementation of the terminal application now mostly consists of the serial driver.

We are now ready to add a new route for the stdin and stdout of processes. To achieve a remote shell, we added `remoted` (not spawned at startup). This application also uses `libterm` but implements a UDP layer to forward input and output over the network. When it first receives a UDP packet, it spawns a new shell process with itself as the terminal server.

We now have a very basic remote shell.

Limitations

While this implementation provides sufficient access to a shell to perform most tasks, the current implementation of `remoted` does not support multiple simultaneous client. This would require some form of multiplexing in the terminal implementation. However, multiple instances of `remoted` can easily be spawned on different ports, allowing multiple remote clients.

To prevent there from being too much overhead in printing over the network, the implementation of `remoted` buffers characters and uses well chosen criteria to flush the buffer every now and then.