Advanced Operating System
Report Group D

Christian Fenia, Philipp Schaad, Jan Müller

December 21, 2017

Contents

Chapter 1

ClownFish

Throughout this project we have tried to keep most of our individual subsystems and libraries as simple as possible. This is to support the mentality that our code should be easy to understand (well ok that didn't always work) and easy to debug and reason about. Keeping things like data structures simple, easy to understand, and general purpose sometimes comes at a cost of performance. We have decided that potential performance gains achieved by over-complicating our system and book-keeping mechanisms would be negligible compared to the additional cost in complexity.

1.1 Memory Management

Our memory manager (`libmm`) is kept relatively simple, such that reasoning about it can be done with ease. In essence each core in our system takes care of memory management by itself, receiving exactly half (in a two core system, less otherwise) of the total physical system memory for itself at boot. Each processor core then has an instance of `libmm` running and manages distribution and tracking of its own segment of physical memory.

1.1.1 Resource Tracking

To track what system resources are free or in use, we employ a very basic doubly linked list, where each element (called a node) has a type (we will get to those in a bit), a capability for the memory region it represents, as well as pointers to the next and previous node together with its region's base address and size in bytes.

We track three distinct types of memory regions with our nodes; free, allocated, and wasted. Free and allocated are fairly self explanatory. The 'wasted' region type is rather rare, but gets used as a workaround for our incomplete capability system. We will go into detail on this in a little bit, but essentially it's a failsafe for when we try to re-allocate a previously used region of memory, since we were unable to correctly revoke that region's capability.

Initially our list only contains one node, which represents the entire physical memory managed by this instance of `libmm`. Upon allocation we split this region up, forking off a region of the size that was requested and shrinking the free node accordingly.

To look for a free region of memory we have implemented a naive algorithm which simply traverses the doubly linked list starting at the head, until a node is reached that has at least the size of the region we are looking for and is marked as free. This gives us a worst case complexity of $O(n)$ when allocating. We can reach this worst case by continuously allocating (rather small) regions of memory without freeing, which implies that our list keeps growing as we fork off new allocated regions from our one contiguous free region at the tail, and we will always have to scan $m + 1$ regions after m allocations.

We decided to use this approach because it is easiest to implement and understand, therefore being less error prone and easier to maintain and upgrade. In addition to that the performance does not get impaired in the standard case, thus making this a viable option. A switch to a binary tree did not seem attractive enough to be worth the additional book-keeping and structure management, which in itself would cause overhead. We have also decided to not perform rearrangement of allocated memory to reduce fragmentation, as that would vastly increase the book-keeping complexity.

1.1.2 Capability Handling

As mentioned in subsection 1.1.1 we initially hold one big node that represents all of our free memory. We section up our device memory into two equally sized pieces in the `'menu.lst.armv7_omap44xx'`, and upon booting each processor's `libmm` gets one of those regions. We represent this region by creating one big ram capability and assigning it to our initial node. This ram capability never changes. Instead we always retype this initial capability, where the new capability represents the offset of the newly allocated region to our original ram capability.

This approach gives us three major advantages: Firstly, the management and tracking of capabilities is easier. If we free the memory backed by one of those retyped capabilities we can simply destroy its representing capability. In addition to that, if we allocate memory we now do not have to worry about the chance that there might already be a retyped capability for this section of memory. Secondly, destroying the memory manager remains a simple task this way, leaving us solely with the responsibility of destroying all capabilities, instead of having to revoke each one of the descendants. It

```
addr allocate(size) {
    ROUND_UP(size, BASE_PAGE_SIZE);
    Node *node = find_node(size);
    if (node->size > size) {
        node->size -= size;
        node = create_and_insert_new_node(size);
    }
    node->type = ALLOCATED;
    err = cap_retype(node->cap, initial_ram_cap, offset, size);
    if (err_is_fail(err)) {
        node->type = WASTED;
        return allocate(size);
    }
    return node->addr;
}
```

Listing 1.1: Allocation Procedure (Pseudocode)

would be harder to keep track of all of those. And finally, with this approach it is easy to add new ram to the manager should new one appear for some reason. All we need to do in that case is to check whether the current big initial capability has run out of free memory, and use the new one instead.

1.1.3 Allocating memory

As mentioned before, our allocation procedure is rather simplistic. We simply start to traverse the list of all memory regions, until we find a match where the requested size fits (the region is greater than or equal to the requested size), and where the region is marked with type free. Note that the size we request to allocate is adjusted to be a multiple of the base page size. If we have found a node that has exactly the size we requested, we simply change its type to allocated. If the node is bigger, we shrink it and add a new node with the size we wanted before it, inserting both back into the list. In both cases we update the capability of our freshly allocated node by retyping the initial ram capability with the correct size and offset.

This is where the node-type `'wasted'` comes into play. Since we cannot correctly revoke capabilities, it can happen that this retype operation fails if the node has previously been allocated and freed. In that case we simply set the type to wasted and recursively re-start the allocation procedure to recover. Listing 1.1 shows a pseudo-code outline of this procedure.

1.1.4 Freeing memory

Freeing memory is quite simple. We revoke and destroy the capability (if it still exists). We set the indicator to free and check if there is a free node right before or after the just freed node. If so, we merge the free spaces together so that we can allocate bigger contiguous spaces of memory.

1.1.5 Slab refills

To refill the slabs, we need memory. To allocate more memory we need slabs. To make sure that we don't run into a deadlock, we made sure that the slab gets refilled when we have less than a few slabs left.

1.2 Memory Mapping

1.2.1 Structures and Bookkeeping

To keep track of the various mapped regions and pieces of memory, as well as the free virtual addresses, we have to keep a significant amount of state for the memory mapping and paging process.

- Free vspace: To prevent address collisions, the free virtual addresses are tracked. This is done using a singly linked list, because its simplicity. This might introduce additional overhead when mapping memory, when the process has run some time and the memory is fragmented. The lookup scales with $0\,(n)$. Currently, freed nodes are simply appended to the list and scale with $0\,(1)$

- Mappings: To be able to unmap a piece of memory again, the mappings are stored. This is done using a singly linked list. A new mapping is added in front of the list. This scales with $0\,(1)$. Finding the right mapping for unmapping memory scales with $0\,(n)$ with the number of mappings in the worst case, when it has to traverse the whole list. However, we think that there are two sorts of mappings. One is short lived and will be unmapped soon after it is mapped. And the others are long living. Our implementation let the long living mappings wander to the back of the list while being fast for unmapping recently added mappings.

- L1 pagetable: a reference to the L1 pagetable is stored.

- L2 pagetables: an array of L2 pagetables. Initially, this list is empty. If a L2 pagetable is used for the first time, it is created and the capref is stored for future uses.

- Spawninfo: When spawning a new domain, we need to copy the caps for the mappings and the L2 pagetables to the new domain. To be able to reuse our mapping code, we keep a reference to the spawninfo that contains a callback function to be used when mapping for a foreign domain.

1.2.2 Mapping

The mapping of a memory frame to the virtual address space of a domain consists of the following steps:

1. If the address is not user chosen, the information about a free block of virtual addresses is computed from with the information stored in the paging state (see subsection 1.2.1).

2. The L2 pagetable corresponding to the virtual address to be mapped is read from the L1 pagetable. If this pagetable does not yet exist, a new L2 pagetable is created.

3. Map the minimum between the number of bytes we have to map and the number of bytes that still fit into the L2 pagetable.

4. Store the reference to the L2 pagetable and the mapping information to be able to unmap the piece of memory again.

5. The steps 2 - 4 are repeated until all memory is mapped. This is the case when the requested size, starting from the virtual address, does not fit into a single L2 pagetable.

1.2.3 Unmapping

Because we stored a fair amount of state, unmapping is easy. All parts of the region to unmap are traversed and unmapped. After this is done, the freed virtual addresses are added to the list of free vspace again (see subsection 1.2.1).

One problem we encountered while implementing the unmapping was that it can be hard to test or demonstrate. Due to compiler optimizations (especially instruction reordering), unmapped memory seemed accessible even after it was unmapped for a short time. **TODO: mention the possibility of pools? was meint ihr? im code haben wir da noch eine bemerkung, dass das sinnvoll sein könnte, damit nicht laufend neue caps erstellt werden müssen. soll ich das noch erwähnen?**

1.3 Demand Paging

1.3.1 Pagefault handling

A pagefault is triggered when virtual memory is accessed, that has not been backed with physical memory. Before handling this exception, we do some sanity checks on the accessed address. We check whether the caller tries to dereference a NULL pointer, if it is trying to access something in the kernel space (above 0x80000000) or if the current stack pointer exceeds the stack. Such a behaviour means misbehaviour or a programming error and the calling thread is killed.

After these checks, the pagefault is handled. We decided to use BASE_PAGE_SIZE (4kB) sized pieces of memory to benefit from address locality and reduce the number of pagefaults. First, the memory is requested from the init domain using a non-malloc path (see subsection 1.3.2). After we received the memory, it is mapped to the caller's virtual address space (see section 1.2).

We did a quick evaluation of other sizes of memory. We run the init domain and let it run our test suite and create 5 threads. This resulted in a total of 18 pagefaults (since the start of the demand paging mechanism). We repeated the test but increased the amount of memory that is mapped by a factor of 2. This resulted in 12 pagefaults, which means a reduction of 33%. Because the overhead of our pagefault handling is no bottleneck to the system and the savings are therefore not that big, we did not implement the larger memory size. The increase of the size would generate additional complexity on the programming side, as we would have to check first, if the next page is already mapped and would need additional testing to make sure that we cover all corner cases.

The stack operates on a separate interrupt stack. We encountered an error, in which the handling of the pagefault would exceed the initial size of 4kB. Increasing the size of the stack to a really large value (16kB) solved that problem.

1.3.2 RAM alloc in pagefault handling

Our initial pagefault handler occasionally caused pagefaults while handling pagefaults, which led to system crashes. We circumvented this behaviour by making sure, that the pagefault handler receives a fully stack based (no memory allocation) path for requesting a new ram capability. The following additional steps were taken, to allow an as direct handling of the pagefault as possible:

- Prevent scheduling of other threads during pagefault handling
- Create a separate lmp_channel for ram cap requests during pagefaults
- Create a separate waitset for the use with the new lmp_channel, to make sure that we give control to the domain providing ram capabilities. (And back, of course)

1.4 RPC

For terminology, our actual remote procedure call system is split into two sections: The RPC system, which is build on lmp_chan's and for transferring messages between processes on the same core, and the URPC system, which is build on a shared memory page and is for transferring messages between the init processes on the two cores.

Now, our biggest regret is that we never unified those two systems properly. An approach at unification which would have involved rewriting aos_rpc to account for that could have likely provided an uniform interface and saved us a lot of pain along the way. But we didn't get there.

The development of the RPC system started out as an adhoc implementation of just adding one RPC call after another and having all the machinery there directly in every single call. This was, naturally, horribly error prone. Also, it was based on a missunderstanding of how to approach this - we registered a send and recv handler on the channel every time we made one of those RPC calls, this turned out to lead to massive issues with multithreading and meant processes could only ever receive if they had just sent. We then had a single shared recv handler, but the memory RPC call needed its own recv handler as that design didn't quite work for that one. "The memory RPC call needs to be handled extra" is a common theme that would repeat throughout our changes to this system.

At first we tried to make some macros and functions while keeping the same basic structure, but that too turned out to be really messy and then it became time for the grand rewrite to the first version of the current system. Instead of init and other processes both doing their own ad-hoc things with the RPC setup and calls, we developed a shared framework for it all.

This framework is set up such that first we automatically grab a fresh ID to mark the logical call we are in so we can associate responses with it, then the memory for the logical call gets persisted and enqueued in the list of things to send. Further, we register ourselves on the channel for sending if nothing is registered there yet. Then when it is time to send it automatically gets a standardized encoding scheme applied to it (the first 32bit word of every physical message gets encoded to contain the type of call, id of the call and length of the total payload of the logical message this is part of) and sends it until it is all gone or an issue occured. Further, the framework also has a standardized receive handler which receives all calls on that channel and recovers the logical message from the physical messages sent, it does so based on the type and id and having a list of which unfinished calls it has where it can look up that type and id pair. Then finally, when a message is fully reassembled we call a process specific (in fact, actually process and channel specific - as we'll see later in the detailed API description). (here an optimisation opportunity for small messages - which are the vast majority - presented itself. Instead of persisting it all and adding it to the list and all those things, we can do it malloc-free by just having that element on the stack and handing it off, as we'll return into this function again and in the normal case free the memory.) While this framework now standardized the encoding of messages - and made it really convenient to work with rpc calls as we will see in the API description - it also presented a large increase in complexity of the code in one particular place (yet still better than the thing before, since before we had a bunch of places that were not as difficult but still difficult and also very labor intensive to keep consistent and we had bug troubles a lot more before the rewrite). The rewrite, however, still had issues. After the first series of minor bugs was fixed, we ran into a pretty big issue: The ram RPC handler now ran over that system, which used mallocs in various places due to its generality and the thus arising need to persist various sizes etc. Our first approach was to do a very hacky thing where we added special conditions for when we were doing the ram_rpc call, using buffers of the structs involved instead of malloc'ing and taking a slightly special path. This, while working for a bit, was naturally doomed to fail as we still got pagefaults at times and the pile of hacks we needed to keep it running kept ever increasing. The next step then was to move it partially out of the system - only receiving as part of it, sending on its own. This cut down on the mallocs further and we had a buffer for what was still needed, which got us a bit further. But the actual solution to this problem came later however, after it had plagued us for weeks with occuring only spuriously and the code around it being annoyingly complex and hard to comprehend - and only getting worse. The solution was, that the ram RPC got moved into its own channel, independently of the others (as at this time, all RPC calls were with init, still) and used entirely its own machinery - very reminiscent of the original implementation in fact. Luckily, on init's side it did not have to be handled differently. However, we still got pagefaults and the occassional strangeness, which later got tracked down to a critical bug in the RPC system's sending - it had only a single queue for sending instead of a per-channel queue, which meant that in particular cases it was possible for messages to not get send to the proper channel or trying to send a message when the channel is not ready but another channel is. This got fixed after a lot of painful investigation as it was really tricky to track down that this was what happened. The current RPC subsystem system is split in two parts - in aos_rpc_shared is (most of) the machinary which was just described.

The most important calls there are shown in Listing 1.2. init_rpc_client creates a new lmp_chan and sets it up, in particular, it creates it with an empty remote_cap, waiting for anyone to connect to it. Also, it uses the handler provided as recv_deal_with_msg as the callback to use when on this channel a logical message is fully reconstructed. The struct recv_list con-

```c
errval_t init_rpc_server(
            void (*recv_deal_with_msg)(struct recv_list *),
            struct lmp_chan *chan);

errval_t init_rpc_client(
            void (*recv_deal_with_msg)(struct recv_list *),
            struct lmp_chan *chan,
            struct capref dest);
```

Listing 1.2: RPC system inits

```c
errval_t send(struct lmp_chan *chan, struct capref cap,
             unsigned char type, size_t payloadsize,
             uintptr_t *payload,
             struct event_closure callback_when_done,
             unsigned char id);
```

Listing 1.3: RPC send

tains the payload (both the data and a capref if any was sent) and its metadata (rpc call type, id, size). This is logically the same as init_rpc_server, except you need to provide it a remote cap to connect to. (In fact, the implementations are currently the same except for the setting of the remote cap in lmp_chan_accept) as well as two for message sending, send and send_response.

Listing 1.3 depicts the primary way to send messages, it does all the explained converting, enqueuing and registering (send_loop does the actual partitioning and sending).

Something that at the time seemed useful was having a callback to be called once the message was fully sent, but in practice our system never evolved to be callback based enough for this to become useful. And something that we should have, but only noticed too late and ran out of time to implement, is a callback to be called in case sending goes wrong, so that we can do custom and message specific handling of issues.

Finally, Listing 1.4 shows the convenience function send_response, which answers a RPC call with the given cap and data. It does a bit of encoding to

```c
errval_t send_response(struct recv_list *rl,
                       struct lmp_chan *chan,
                       struct capref cap,
                       size_t payloadsize,
                       void *payload);
```

Listing 1.4: RPC send_response

```
rpc_framework(void (*inst_recv_handling)(void *arg1,
                  struct recv_list *data),
              void *recv_handling_arg1,
              unsigned char type,
              struct lmp_chan *chan,
              struct capref cap,
              size_t payloadsize,
              uintptr_t *payload,
              struct event_closure
                  callback_when_done)
```

Listing 1.5: rpc_framework

make sure that the receiver gets the ID of its original RPC call in an expected place, but sadly at the cost of another level of persisting which currently is implemented rather inefficiently (a full copy of the data). If we had the time, this could be rewritten in a way which doesn't require that, for example by adding an extra field to the recv_list where it can get persisted in or something along those lines. The real gain, however, was not in having those functions, but that they allowed us to write the rpc_framework function (shown in Listing 1.5), which looks like a really wild one - and it is. It is in aos_rpc and works in conjunction with a logical message recv handler which is set up in such a way that it takes the type and id info we were given and looks it up in the list of calls that we made which we are waiting for an answer on.

Now, what rpc_framework does is that in the first two parameters, it takes the function which handles the response to the call we are about to make - as well as some state for it (arg1 we provide here, data we will get provided). And the other parameters are just all the info we need to make a call. Then it registers us that we want to get a response, sends out the message and pumps the waitset until we have received an answer (thus being blocking). Which means all we now needed to do to add another RPC call was to implement a very simple recv handler (which usually just deserialized and copied the data from the thing we got in response into the thing we stored as arg1) and serialize our data for sending.

Listing 1.6 shows an example for a call for getting the nameserver's cap (so we can connect with all the other services by asking the nameserver for those).

So, overall, the RPC system had a great API that was really simple to use and didn't need a deep understanding of how lmp_chan's work. However, this was paid for in bugs and complexity, but in the end it was a worthwhile tradeoff as we could rapidly extend this as needed.

Our one big complaint is that we never made it backend agnostic to

```
static void get_nameserver_recv_handler(
    void *arg1, struct recv_list *data)
{
    struct capref *retcap = (struct capref *) arg1;
    *retcap = data->cap;
}

errval_t aos_rpc_get_nameserver(
    struct aos_rpc *rpc, struct capref *retcap)
{
    rpc_framework(get_nameserver_recv_handler,
                  (void *) retcap,
                  RPC_TYPE_GET_NAME_SERVER,
                  &rpc->chan, NULL_CAP,
                  0, NULL, NULL_EVENT_CLOSURE);
    return SYS_ERR_OK;
}
```

Listing 1.6: aos_rpc_get with helper function

integrate the cross-core communication system (dubbed URPC) into it.

1.5 URPC

As we do not have a fully unified system we have the init<->init cross core communication bit which is dubbed URPC.

Originally it was just a single page split into two 2 roughly 2kb section (slightly less as we had a smaller shared head at the top of the page) and could just transfer messages in 2kb blogs and signalling the other side that the message arrived.

This got adapted when the actual cross core communication chapter came up and said it should be done in cache lines (or fake cache lines, rather). So we adapted it to be chunked and used that opportunity to adopt it to a similiar framework system as the RPC one - again, sadly we missed the opportunity to unify it at this point and then never got to it as it continued to be more and more work to actually do so.

As the URPC system does not involve the kernel it turned out to be a much simpler system and way less buggy (impl of the actual meat of the system is found in urpc2.c). We segmented the page into 64byte chunks, which had 1 byte for flags and 63 bytes for data and we kept track of them as a ringbuffer. We learned from the RPC system here and did a better encoding - in part aided by this system only having to deal with a single "channel" and as we will later see a FIFO queue of messages - we only send the length and the metadata once. The flag byte is primarily used to signify if this entry is empty or written to, as well as what encoding we will be using (to parse the size correctly, this could have been extracted out but we had flags left, so it was easier this way). We also used flags to signify the start and end of a message but, in retrospect that was unnecessary.

A neat little thing is that because we send in a loop we only need a single memory barrier (conveniently made easy and clear as MEM_BARRIER macro) at the start of the send (and also the recv loop, which is just the inverse of the send loop, really) as well as after the loop for when we exit the loop due to the queue being full (or empty, in the recv's case), or the message fully send (or received).

Ontop of these two core loops we run an endless loop which checks if can (and have something to) send (or receive) (here we learned it's crucial to thread_yield() when we have nothing to send/receive, or the queue is full/empty, to achieve good performance).

This means the thread ever only does that. Ontop of that is a somewhat familiar machinary - we can enqueue messages and register receive handlers. The enqueuing is done from another thread, into a shared threadsafe queue.

For implementing the actual URPC calls we first considered making an RPC equivalent for every single URPC call, but then instead added some more machinary - which while fairly ugly code and being a bit hacky due to, once more, time constraints - allowed us to root an arbitrary RPC call to init over URPC, thus allowing the two init processes to act as if they were a single one for the purposes of RPC.

An init process just receives a fake call to its rpc receive handler with a normal recv_list struct, except that the channel is set to NULL - which can never happen in a normal RPC call and is thus a fairly obvious and simple distinguisher of the source.

In theory, this same system could be used to allow an arbitrary pair of processes to communicate directly with each other, if they can be arranged to share memory. This, however, we never got around to implementing and instead just implemented a proxy service using the init processes for communication. Which has the benefit of not having the extra looping thread overhead and is easier to set up than sharing physical memory across cores (since that'd require us to mint a new ram cap and if we want to properly track it we'd need to extend the memory manager to somehow deal with memory shared across cores. This is in particular a problem as we can not revoke capabilities and thus can not reave the memory when one of the processes dies.)

1.6 Spawning Processes

Our spawning code is nothing fancy and pretty much follows the procedure outlined by the book. Let's go over the individual steps.

Since we do not have a file-system in our OS, we are dealt a lucky hand in this portion. All we have to do to find the correct module is look up the name in the multiboot image. Since the call to `spawn_load_by_name` of our spawn library can (but does not have to) contain an arbitrary number of arguments, the first thing we do is parse the requested name, filtering out the actual program name and separating the arguments away. We now look up that name in the multiboot image. Once we have found our module, we identify the frame it sits in and map it into our own address space with `paging_map_frame`, which gives us a virtual address pointing to our ELF file.

In the next step we need to set up a bunch of capabilities for our new process. The first one would be an L1 CNode, with and into which we then create a bunch of L2 CNodes using `cnode_create_foreign_l2` (since we have already created the L1 node). We map the TASK CNode (which contains information about the process) into the child process's capability space, and then back all it's L2 CNodes with some (exactly a page) of memory.

We now have to set up the child process's virtual address space by creating an L1 page-table in our current virtual address space, which we then copy in to the child process's L1-page-table slot. This step is followed by calling `paging_init_state`, which can now set up the paging state of our new process.

We are now ready to load the ELF binary and initialize the dispatcher that will take care of launching our process. This is a rather tedious process of just creating the dispatcher and an endpoint to it and then copying everything into the child process's virtual address space. In there, we then fill in the dispatcher struct with details like where the GOT sits, what the process's domain ID will be initially and what core it will be run on.

All that remains to be done now before we invoke the dispatcher, which will start our process, is to set its arguments correctly. In this step we will check if we have previously received any arguments from the user. If so, we will map those into the child process, if not, we will look up if the multiboot image provides some arguments and use those instead. After this we are done and are ready to run our process by invoking the dispatcher.

Chapter 2

Individual Projects

2.1 TurtleBack (Shell)

The shell is a very essential part of any OS, as it allows the user to interact with the system at its most basic level. However, for the shell to make any sense to begin with, one first needs a way to talk with the user. This is typically done with a standard text interface. Our system thus needs to have the ability of printing text to the screen, and read text back that has been typed in by the user. It is best to do this reading and writing as centralized as possible, and for this we will first look at the subsystem responsible for that bidirectional input and output; the terminal driver.

2.1.1 Terminal Driver

Our terminal driver is kept as simple and protocol free as possible. For this reason it is running in the `init` domain on both cores. Let's first discuss what this implies for our output to the screen.

When printing to the screen with functions from the C standard library, the system first checks whether we are in the `init` domain or not. If we are indeed, we directly talk to the terminal driver, requesting to have it print to the screen for us. The terminal driver (since it is inside of the `init` domain) can do so via system call. If we are however in a different domain, we will send out an RPC to `init`, which in turn uses the terminal driver to service our request and print to the console via system call. This can happen on both cores simultaneously and entirely independently, which keeps the system's complexity at a minimum.

Input is a little less trivial, but still very simple. Since we have two instances of our terminal driver running (one on each core), we will call the one on core 0 our master and the one on core 1 our slave instance. Since we can only have at most one instance registered for interrupts from the UART interface, we will let the master instance take care of interrupt handling, and hence register it to receive UART interrupts. If an interrupt is received by the master instance, it will perform a system call to retrieve the typed in character from the UART buffer and store it in its own input buffer (which we will get to in a second). Since the slave instance will not receive that interrupt and does not have the ability to retrieve that same character from the UART again, the master instance performs a URPC with which it sends the received character to the slave instance, which can add it to its own buffer.

Input Buffers

Our terminal driver supports two separate modes of buffering input from the UART interface. Line- and direct-mode.

In line-mode we buffer - as the name suggests - entire lines of text. Our line buffer consists of two separate character arrays, one of which is considered to be our write-buffer and the other one is the read-buffer. When a new character arrives (via interrupt if we are on core 0, via URPC otherwise), we will first perform a little bit of preprocessing with it. The most obvious step is to check if it is a character or signal indicating the end of a line (EOT, LF, CR). In that case we null terminate our write-buffer and switch the two buffers. So now when someone tries to read from the read-buffer, it will get a character from the newly finished line, and we will start writing to a new (now invalid and thus considered to be empty) line. The remaining preprocessing steps are less important and mainly matter for the second point where the two modes differ from each other; echoing. In line-mode let the terminal driver manage echoing of newly arriving characters, and not whatever process is reading that input. This is also why we do not use this mode for the shell, but instead use our second mode of operation.

In direct-mode the steps performed by the terminal driver are kept to a bare minimum. Our buffer has the form of a ring-buffer (FIFO), and whenever a new character arrives, we simply add it to the buffer and advance the tail by one, without really caring about what the character value was. This gives the mode its name, as we will directly push ALL input forward to the consuming process(es), without performing any cleanup or echoing the characters back to the user. Our consumer has to take care of that.

In both of those modes we simply discard new characters if our buffer is already full. In addition to that, escape and control sequences that make a mess or might be important for the consumer are analyzed by the terminal driver and stored as one special character (instead of their raw form, which typically is 2-3 characters). This can be easily done, since the ASCII standard for our supported symbols only consumes 128 values. A character (byte) can hold 256 values, so we can use the remaining spaces to encode 128 custom characters. For example left-, up-, right-, and down-arrow keys will get escaped by the terminal driver and stored as special values, so the consuming process understands that they were pressed.

When a process performs an input operation using one of the C standard library functions, we differentiate between `init` domain and other domains again. In the case of `init`, we directly query the terminal driver for new characters. Other domains will perform an RPC to init, which will serve the character via the terminal driver. In both cases, the call to the terminal driver will block for as long as the input-buffer does not have any ready characters. As soon as characters arrive, one is returned and the consumer process can analyze it. Which input-buffer will get queried depends on the currently selected mode in the terminal driver. To make sure the two driver instances' input-buffers are in sync, whenever a character is removed from one of the two instances, it will fire a URPC to the other instance, moving the head by one space, thus also removing the character.

2.1.2 The Shell

Our shell is a separate process (`turtleback`) that can be spawned on either core once the system has completed its boot process (core 0 by default). The main shell process in itself is not very interesting. All its doing is performing an endless loop of reading characters from the terminal (via C standard library functions, which go over `init`'s terminal driver), storing them in a buffer and then parsing that line once the user hits a new line. The interesting part lies in just the parsing of that input line and what that sets in motion.

The shell serves two main functionalities. It presents the user with a set of built in functions - with the unspectacular name 'builtins' - that can be used to interact with the operating system, and it will spawn other processes for the user when he types their name into the console. Let us first examine the built in functions that our shell provides.

Builtins

- `help` - This is probably the most basic builtin that any shell (or application for that matter) should have. It does exactly what the name says; it prints out a list of all available builtin functions and offer a helpful little explanation of what each of them does. Supplying it with the name of another builtin will print more information about how to use said function.

- `clear` - Another relatively self explanatory function, found in pretty much every command-line based interface; it will clear the screen for you (preserving the scroll buffer). This is done by printing the ANSI sequence ESC - [- 2J to the terminal.

- `echo` - Also one of the functions everyone knows; this will simply print whatever arguments you supply it with back out onto a new line in the terminal. It will print all 'words' (arguments) spaced out with one single space, no matter how far apart the arguments were supplied (yes, it behaves exactly like bash's `echo`).

- `ps` - Like its bash equivalent, `ps` will simply perform an RPC to the process manager (in the init domain), which in turn will print a list of all running processes, together with their process ID and which core they are running on.

- `time` - This is the last one of the commands that behaves pretty much like bash; `time` must be used as a prefix command to any other builtin or binary (with arguments or not). It simply times how long it takes for the execution of said command/program to complete. This is done by getting the clock-cycle count before and after the execution, calculating the difference and dividing it by the CPU clock frequency, which gives us the exact time in seconds and milliseconds.

- `led` - A new, but rather unspectacular command, that simply tells the `init` domain to toggle the D2 LED via RPC. If the LED was on it gets turned off, and vice versa. Toggle. That's all. Yes.

- `testsuite` - This performs a little RPC to the `init` domain, which will run a standard test suite that is built in to the OS and contains a couple of memory and regression tests that were used during the development of this operating system. It only exists because we can.

- `memtest` - The memory test will take a single argument (a positive number), that tells it how many bytes the user wants to test. It will then allocate a contiguous region of memory of that size and test it for read- and write-errors.

- `threads` - This is simply a demo of user level threads. It takes a single positive integer argument and will spawn a number of threads equal to that integer argument. Those threads simply print out the line `"Hello world from thread X"`, where `X` is its thread-id.

- `oncore` - Serving as another one of those prefix commands (like `time`), `oncore` will take the program spawning command it prefixes and spawn said binary on the core specified. So if we type in `'oncore 1 hello args..'`, this will spawn the binary `'hello'` on core 1 with arguments `'args..'`. This is done with a standard spawn-process RPC to init, which can already handle cross-core spawns.

- `detached` - This is the final prefix command like `time`. Like `oncore`, it serves as a prefix purely for spawning programs, not shell builtins. Its purpose is pretty simple; `'detached progname args..'` spawns the program `'progname'` with arguments `'args..'`, but will not wait for it to finish before returning to a new shell prompt and listening for user input.

Parsing User Input

We parse the user's input every time he or she hits the end of a line and starts a new one (typically by pressing `Ctrl + J`, `Ctrl + D`, or `Return`). The line of text in the buffer gets split up into a series of token. Each space in the line represents the end of a token and the start of the next one, whereby multiple consecutive spaces get dropped and will count as a single token separator. We now look at the first token and treat it as our command, thus deciding what to do based on its value.

We first check if the user wants to execute one of the builtins TurtleBack provides. Our shell holds an internal list of all available built-in functions, together with a handler function attached to each one of them. We can thus simply iterate over this (relatively short) list, performing string matching against each of the identifying commands. If we have a positive match, we will execute that match's handler function, passing all but the first token along as its arguments. If we hit the end of the list without a match, we move on, treating the entered command as a binary the user is trying to spawn.

This leads us to how we handle process spawning. We use an UNIX-like approach, where a program can be launched simply by typing the name of its binary into the console, without a special run command. Internally this will launch an RPC to the `init` process, where we look up if the queried binary name can be found in our multiboot image (since we do not support a file-system, that's the only place we can spawn from). `init` then launches the process (again, passing all the remaining tokens along as arguments to the new program), registering it in our process manager, and returns the PID it just assigned to the process. TurtleBack then queries `init` (via RPC) for the existence of that process, blocking until that PID could not be found anymore. This ensures that we await the termination of our spawned program before showing a new shell prompt and accepting user input, thus allowing the spawned process to take control.

If the requested binary could not be found, the PID returned will have a special value (UINT32_MAX), which tells TurtleBack that we still have not been able to handle the user's command. At this point we know that if we were unable to serve the user's request, he or she must have entered a wrong command, so we echo the command back (just the first token) and tell the user that this does not match to any supported command.

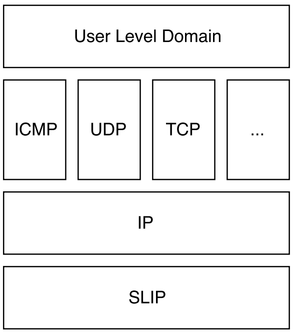


Figure 2.1: Code structured according to the OSI reference model.

2.2 Network

2.2.1 Overview

The implemented network stack is organised on a code level according to the OSI reference layer. However, on the running system, only the transport layer is running isolated from the others. An overview over the implementation can be seen in figure 2.2. The receiving side of the slip protocol, the IP protocol and the transport protocols (UDP, ICMP) are running on the same thread to reduce the overhead introduced by scheduling or the even larger overhead of a call to the RPC system. Another advantage of this system is, that there is less possibility of an internal buffer overflow.

For the communication between different network related domains, an internal network messaging protocol (INMP) was implemented. This protocol relies on the underlying RPC/URPC implementation. This allows for multiple user level domains accessing the network, for example the implemented udp_echo and udp_terminal domains. There is one master domain called network, which starts the network logic. This process takes the own IP address as input. Currently, it is not possible, to change the IP address after this initial setup.

Currently, there are two drawbacks: Because the nameserver is used to connect the network domains and does not support cross core lookups, all network tasks have to run on same core. The other drawback is, that it is not possible to have multiple IP addresses, currently. If one tries to run a second network stack on the other core, the behaviour is not defined as the interrupts are not guaranteed to work as intended anymore.

2.2.2 SLIP

Receiving

The serial input is fed into a circular buffer. A thread implementing the SLIP protocol reads from this buffer byte by byte and does the necessary substi-

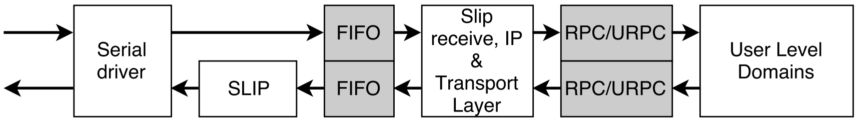


Figure 2.2: The way of a packet from entering on the serial line until leaving the system again. White boxes correspond to individual domains or threads. Gray boxes are used to illustrate the communication channels

tutions given by the protocol. Once the message is complete, it is written into a single packet sized buffer. This might lead to problems if the packets are incoming at a higher rate than they are processed, but this behaviour was not yet encountered and the buffer size could easily be increased. The processing should be fast, because it consists only in local comparisons and the reassembly of packet headers for incoming ICMP echo requests. UDP packets are immediately sent to the handler of the corresponding port. A potential overload would show itself in a overflow of the circular buffer which would cause packets to be dropped because of a wrong ip checksum.

Sending

When sending, the behaviour is similar. The packet to be sent is read from a small buffer. The substitutions needed for the SLIP protocol are made and the resulting stream of data is written to the serial port byte by byte.

2.2.3 IP

Receiving

When the IP handler receives a packet from the SLIP handler, it does some sanity checks first:

- Checksum check: check the integrity of the received packet.
- Version check: check that the incoming packet is using IPv4.
- Length check: make sure that the packet and the header are at least as long as the minimal header length.
- Fragmentation check: check for fragmentations (fragmentation flag sequence ID set)
- Destination check: check if the device is really the destination of the packet (local IP address matches the destination IP address)

If any of these tests fail, the packet is dropped.
If the packet passes the tests, the byte order of the source IP address is changed from network byte order to little-endian and is delivered together

with the payload to the right protocol. Currently, there are the ICMP and UDP protocols supported. Again, if the packet uses another protocol, the message is dropped.
Because the os is not meant to be used as a routing device, the hop count is not reduced or checked on incoming packets.

Sending

When the IP handler receives a packet, a standard set of headers are set and the checksum is computed. After the IP packet is arranged, it is handed to the SLIP handler. At the moment, there is no possibility for a domain sending a packet to change the headers of the IP packet (besides the destination address of course). We apply a standard set of headers to all outgoing packets.

2.2.4 ICMP

The handling of ICMP is simplistic. Currently, the only supported function is the echo reply protocol, which listens to an incoming echo request and replies with a echo reply message. Other incoming ICMP based protocols are dropped.
The ICMP handler offers a function which allows any user level domain to send arbitrary ICMP packets. However, there is currently no application making use of this functionality.

2.2.5 UDP

Open new port

All UDP ports are implemented by user level domains. A domain can open a new port by sending the appropriate message using the INMP. The domain has to implement the handler for the incoming messages and can then send a request to open a new port to the UDP handler. The handler will check if the port exists and if not, add it to a singly linked list and direct incoming datagrams on that port to the specified domain (again using the internal network message protocol). An open port can be closed again. Currently, there is no check on who is trying to close a port. An obvious improvement would be, to only allow the domain that opened the port and some monitor domain to close the ports. Also, there is currently no check, if the receiving handler of the port is still running or has crashed or exited without closing the port.

Receiving

When the UDP handler receives a packet, it UDP handler checks the destination port and forwards the packet to the right user level domain using the INMP. If the port number is not known, the packet is dropped. The incoming and the outgoing port, as well as the source ip address and the payload is sent to the handling domain using the internal network messaging protocol.

Sending

Using the internal network messaging protocol, a user level domain can send a packet to the UDP handler which writes the header of the datagram and forwards it to the IP handler. No checksum is created, because according to the RFC on UDP, the UDP checksum is optional.[1]

2.2.6 User Space Applications

Two user space applications were implemented. An UDP echo server was implemented, that automatically connects to the network process on the same core using the nameserver. All this process does, is to fetch the user input and return it to the user.
The UDP terminal server is more interesting. This application is a minimal shell, that allows the user to spawn processes and returns the output of that process to the user. It is currently not allowed to start the TurtleBack shell, as the input integration is not done. The way this works, is that the network terminal app passes a special flag to a spawning process, that signals that it should report to the network terminal. The new process will then use the nameserver to redirect its output to the network terminal. This system could be improved by integrating it to the TurtleBack shell. From a security point of view, a login mechanism should be added.

2.2.7 Performance

TODO: Measure max bandwidth

Ping

Two tests were performed to measure the performance of the implemented ping-reply mechanism. Both tests consisted of sending pings with 31 different payload sizes. The tests were run 100 times for each size. The Linux ping utility was used to perform both tests.
The first test checked the reliability of the system. For a single payload size, we send 100 packets and note, how many replies we receive. The amount of

received ping replies is shown in figure 2.3a. We note, that for sizes that up to about 1.5kB, at least 98% of the ping packets generate an answer from the system. Note that on sizes larger than 1kB, some of the icmp echo-reply answers contain a faulty payload. This means, that the message sent and received do not have the same payload. However, only less than 3% of the packets are affected. We see, that there were no replies of the system for payloads larger than 1.5kB. The reason is, that the packets are dropped by the tunsip utility. The error code of -7 indicates a ICMP checksum error. The error happens on the host side, before the packets reach the device under test.
In the second test, the actual RTT time for the differently sized ping requests were measured. Sizes larger than 1.5kB were excluded from the test, because of the 0 yield. The mean, as well as the lowest and the highest values for the RTTs can be seen in figure 2.3b. Different things can be noted from the data collected:

- The mean RTT is linearly increasing from 0 up to about 800B of payload. **TODO: what does this tell us?**

- At around 1kB of payload, the system performed the worst. **TODO: find out why**

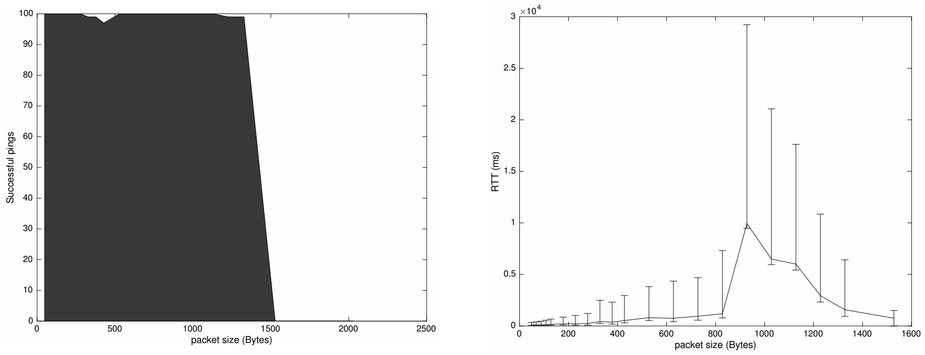
- **TODO: write more**

- larger peaks
- reduced RTT after about 950 B

115200 14400 Bps 14kBps 1.2 ms/B increase

UDP echo

Using nmap, the latency of the response of the echo server was measured. The test was repeated 1000 times. The following values were measured:
maximum latency: 230 ms
minimum latency: 47 ms
mean latency: 68.5 ms
the measured values had a standard deviation of 21.7 ms. **TODO: comment on this**



(a) Successful pings (out of 100) for different ping payloads

(b) Round-trip Time (RTT) for different ping payloads

Figure 2.3: Ping tests performed using 31 different sized ping request packets.

Bibliography

[1] J. Postel. User datagram protocol. STD 6, RFC Editor, August 1980. http://www.rfc-editor.org/rfc/rfc768.txt.