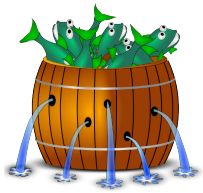

ADVANCED OPERATING SYSTEMS
Milestone 1: Physical Memory Management
Fall Term 2016

Assigned on: **30.09.2016** Due by: **07.10.2016**

In the first milestone, you will implement the infrastructure for managing (physical) memory. You will also see and use *capabilities* for the first time as a mechanism to identify regions of physical memory – the basics were covered in the last lectures. The reading material for capabilities is on the course webpage.

The branch (week2) that you merge with your tree will contain a more complete cpu driver (i.e., kernel) and a very basic version of the first user-space program called init as well as a stub library called `mm` in the `lib` folder which you will have to implement.

This milestone is the first step towards a (hopefully) complete memory system at the end of this course. The goal for this week is to perform memory *allocation* and basic memory *mapping* functionality within the init address space. Note that in the forthcoming weeks you will continuously refine and extend your memory management system with more functionality. Therefore, it is important that you write correct and understandable code to form a solid basis for future assignments.

Step 1: Get Prepared

For this milestone you will get a full Barrelfish CPU driver and a somewhat more complete `init` process, as well as some support libraries (`libc`, an initial version of the library OS `libaos` that you will use and extend, and `libmdb` which is used by the kernel to track capabilities) as well as a bunch of helper libraries you may find useful.

The easiest way to get the new hand-out code is by merging the `week2` branch with your current master branch:

```
$ git checkout master
$ git merge week2
```

In this milestone, you will build the full Barrelfish CPU driver, as well as your first user-space program (init). Therefore, the commands have slightly changed. First, create a build directory:

```
$ mkdir -p /local/<your nethz-name>/build_milestone_1
$ cd /local/<your nethz-name>/build_milestone_1
$ <path to source tree>/hake/hake.sh -s <path to source tree> -a armv7
```

Next, build the tree:

```
$ make -j7 PandaboardES
```

Finally, boot the created image on the Pandaboard:

```
$ make usbboot_panda
```

An unmodified source tree should give you the following output:

```
kernel ARMv7-A: Trying to enable interrupts
kernel ARMv7-A: Done enabling interrupts
kernel ARMv7-A: Calling dispatch from arm_kernel_startup, start address is=204000
init.0.0: paging_init
init.0.0: init: on core 0 invoked as: init 2097152
ERROR: init.0 in initialize_ram_alloc() ../usr/init/mem_alloc.c:57
ERROR: Can't initalize the memory manager.
Failure: (  libbarrelfish) functionality not implemented yet [LIB_ERR_NOT_IMPLEMENTED]
Aborted
revoking dispatcher failed in _Exit, spinning!
```

EXTRA CHALLENGE

From this point on in the course, you'll be using the regular Barrelfish kernel (or CPU driver, as we call it). We'll offer extra points for every bug that you can find *and fix* in our code!

About Capabilities

In this assignment you will use Barrelfish's capability system to manage physical memory. The capability system is Barrelfish's way of tracking and authorizing access to memory (and, as we will see, many other physical resources in the system) to a user program.

Think of capability as a special kind of handle or pointer. Capabilities cannot be forged, but can be passed between applications. There are different types of capabilities for different types of memory, such as page tables or user-accessible (mapped) memory.

You will encounter the following types in this weeks assignment (see `capabilities/caps.hl` for a full list of Barrelfish's capability types):

- `ObjType_RAM` (for untyped, physical memory)
- `ObjType_L1CNode` (for maintaining information about capabilities itself)
- `ObjType_L2CNode` (for maintaining information about capabilities itself)
- `ObjType_Frame` (for pages)
- `ObjType_VNode_ARM_l2` (for ARM level 2 page tables)

Barrelfish stores a process' capabilities in *slots* in that process' capability space (CSpace). Each CSpace has a hierarchical structure (similar to a two-level page-table), and is constructed on demand by retyping ram capabilities into `L1CNode` and `L2CNode` types. These CNode regions are only accessible by the cpu driver, however the user-space program is responsible for allocating them on the cpu driver's behalf.

There are also a number of functions that allow you to request new capabilities of these two types. You will likely encounter and use

```
cnode_create_l1(struct capref *ret_dest, struct cnoderef *cnoderef)
cnode_create_l2(struct capref *ret_dest, struct cnoderef *cnoderef)
```

to allocate level-one and level-two CNodes,

```
frame_alloc(struct capref *ret, size_t bytes, size_t *retsize)
```

to allocate Frame capabilities that can be mapped into an address space and

```
arml2_alloc(struct capref *ret)
```

to allocate capabilities for ARM level 2 page tables.

All these functions will rely on the RAM allocator function

```
ram_alloc(struct capref *ret, size_t bits);
```

to allocate untyped, physical memory first and then retype that area into more a more specialized type by calling `cap_retype` on the respective capability. The invocation to retype a capability looks like this:

```
cap_retype(struct capref dest_start, struct capref src, gensize_t offset,
           enum objtype new_type, gensize_t objsize, size_t count)
```

As we can see, the retype operation is fairly generic and can not just change the type of an existing capability (or region) but also modify the size or work only on a part of the source capability.

The first part of this exercise is about implementing the necessary infrastructure for `ram_alloc` which will then give you the means to allocate all the other objects as well.

The `struct capref` type is what we use to reference a capability slot (and the capability in it) in user space. It uniquely identifies one address (i.e. one slot) in the process' capability space. It can be understood as a pointer to a region inside an `L2CNode` memory area that stores information about that particular capability.

We need to make sure that applications cannot construct arbitrary page tables — instead, we must ensure that an application can only put an entry in a page table if it refers to a physical frame to which it already has authority.

To enforce this, page tables are updated via the capability system. We invoke capabilities as if they were objects — essentially calling a function associated with the capability's type — and we use these invocations to (among other things) modify page table entries. The high level interface for writing a page table entry looks like this:

```
vnode_map(struct capref dest, struct capref src, capaddr_t slot,
          uint64_t attr, uint64_t off, uint64_t pte_count)
```

The `slot` argument gives the page table entry to modify, `attr` are the mapping attributes (you probably want `VREGION_FLAGS_READ_WRITE`), and `off` is an offset into the `src` memory region. The last parameter is there to indicate the number of pages you'd like to map if you're mapping a Frame capability of more than 4k in size.

Also, the Barrelfish CPU driver enforces that each copy of a capability is mapped at most once, so you will have to use

```
cap_copy(struct capref dest, struct capref src)
```

to create copies if you're partially mapping large frames.

You'll notice that `vnode_map()` looks a bit generic, even though you'll only be using it for a couple of cases. One reason is that we use the same function to map L2 page tables in the L1 page table and pages in the L2 pagetables. You will see the other reason for this later in the course — Barrelfish can actually construct page tables for *any* processor architecture this way, even on a different architecture.

Barrelfish stores the capability to the L1 pagetable at a well-defined location in the capability space, namely slot zero of the `page cnode`. You can create a `struct capref` to that location by hand as follows:

```
struct capref l1_pagetable = {
    .cnode = cnode_page,
    .slot  = 0,
};
```

and use that `struct capref` as the destination parameter for `vnode_map()`. `cnode_page` is defined in `lib/aos/capabilities.c`.

Step 1: Implementing a physical memory manager

In the first part of this exercise, you will implement `libmm`, a library to manage memory (i.e., RAM capabilities). The library will form the basis for dynamically provisioning memory resources to various applications running on your system. In particular, you need to implement the various stub functions in `lib/mm/mm.c`.

As we've already seen in the lecture, the Barrelfish model delegates memory management to user-space applications (e.g., `init` in our case). Initially, the kernel will query the underlying hardware to figure out which address ranges are backed by physical memory. From this information it will construct a series of RAM capabilities which it provides to `init` on start-up. Study the `initialize_ram_alloc` function in `usr/init/mem_alloc.c`. You will find that it walks through all the initial RAM capabilities as provided by the CPU driver and tries to add them to your memory manager using `mm_add`.

Afterwards, the line `ram_alloc_set(aos_ram_alloc_aligned)` will make sure that from now on, any request to allocate a region of memory will make use of your implementation by using `mm_alloc_aligned`. You will find that the initial RAM capabilities provided by the kernel are quite large (typically multiple mega-bytes), whereas the typical object size to be allocated is much smaller (e.g., 4 KiB for individual pages or 16 KiB for an L2CNode). Luckily, capabilities can be split into smaller chunks using the `cap_retype` invocation. You must make sure that your memory manager does not introduce unnecessary fragmentation by splitting capabilities into smaller regions first.

A side-effect of splitting capabilities is that you will now require additional slots for storing the newly created capabilities in your CSpace. Also, your memory manager will likely require additional meta-data to track the state of free capabilities. For this reason, we will provide you with a *slab* and a *slot* allocator. Study the code of these two allocators in `lib/mm/slot_alloc.c` and `lib/aos/slab.c`. The slab allocator is a simpler version of malloc that only allocates objects of a fixed size and needs to be refilled manually (for example by using `slab_default_refill`) in case it is out of memory. Initially, you can fill it simply with a static buffer that is big enough. Once you have completed the next step (mapping frame capabilities), you should change that and refill it with dynamically allocated frames in case it runs out of memory. The slot allocator is responsible for allocating space that holds the capability meta-data as required for use by the kernel. The slot allocator can also run out of space, so you need to make sure it always has enough slots left by calling the slot refill function.

Step 2: Mapping frame capabilities

Once you have completed an initial version of your memory manager, it is time to write a first basic mapping function for frame capabilities. In the following milestones you will build a complete virtual address space management system, but for now it suffices that you are able to map a frame capability at a free location in your virtual address space.

Implement the `paging_map_fixed_attr` in `lib/aos/paging.c`. With a working version of `lib/mm`, you can now allocate `ObjType_VNode_ARM_l2` capabilities as well as frame capabilities and use `vnode_map` to construct the virtual address space.

For simplicity, this week you can still assume that the frame you are trying to map always fits inside a single L2 page-table and the virtual address is chosen such that it does not overlap (i.e., you do not have to modify two L2 page-tables in one mapping). This will minimize the amount of meta-data you have to track for your initial mapping implementation. But, note that already in the next milestone those assumptions likely no longer hold.

Note: you should have all the state of your implementation in the `struct paging_state` of which you'll find an instance called `current` setup as a global variable in `lib/aos/paging.c`. Also, have a look at the different comments that are marked TODO in `lib/aos/paging.c` to see what types of functionality you will need in later exercises.

After you have verified that you can map frame capabilities in your address space, you can now implement `slab_refill_pages` in `lib/aos/slab.c`. This will add the necessary functionality in the existing slab allocator to refill it dynamically if it ever runs out of slab space. And it will serve as a first test for your frame mapping functionality. You should have all the necessary mechanisms to implement `slab_refill_pages` by using a combination of `frame_alloc`, `paging_map_fixed_attr` (for now use a manually chosen VA offset), and `slab_grow`.

Once completed, should go back and verify that your code works by writing a test that checks if you correctly make use of the provided slot and slab allocators in your `libmm` implementation. For the slot allocator, you can check that your code can handle more than 256 allocation. For the slab allocator you can similarly try to exhaust the initial static buffer space to force a refill (make sure the slab allocator will use your `slab_refill_pages` function).

THE MILESTONE

- Implement `libmm` to manage physical memory.
- Show that your implementation can allocate and free RAM capabilities.
- Show that you can map a frame capability in your address space.

CHALLENGES

- Handle running out of capability slots in `libmm` (allocate more!).
- Handle running out of slab space in `libmm` (allocate more!).

ASSESSMENT

- Tested allocation and reclamation of ram capabilities in `init` for various sizes and allocation patterns.
- Be able to explain the design of your memory manager as well as any the provided code you have used (i.e., slab and slot allocators).
- Show that you can map and write memory in `init` and be able to explain the code you have written to do that.