ADVANCED OPERATING SYSTEMS
Milestone 5: Booting and using the second core
Fall Term 2016

Assigned on: **11.10.2016** Due by: **18.10.2016**

1 Overview

In this milestone, you will bring up the second core, establish (simple) inter-core communication, and run applications on the second core.

The work for this milestone consists of two principal tasks:

a) *Boot the second core*
Bringing up a core on Barrelfish means booting another copy of the CPU driver. To do this, you will use *Coreboot*, as described in Section 2.

b) *Manage memory across cores*
Each core's CPU driver is completely independent. You must explicitly manage the distribution of resources (RAM) between your two CPU drivers. To do so, you can use the *kernel capability* to forge new capabilities, as described in Section 3.

2 Bringing up a Second Core

This step concentrates on bringing up the second core and running code on it.

Here is a brief overview of how the bootstrapping process for the second core works: on first booting, each core checks its core ID (a hardware register). Core 0 will continue as the *BSP* core — the one that you've used so far. All other cores go to sleep (using `wfe`), and wait for the BSP core to wake them, by sending an *event* (instruction `sev`).

The BSP core places a *boot record* in a well-known location, including the address of a `core_data` structure, which provides all necessary initialisation data for the new CPU driver, and the ID of the core that should boot (`target_mpid`). Every core wakes up together on an `sev`, and checks `target_mpid` against its own core ID. If it matches, it jumps into the CPU driver (at `cpu_start`). Otherwise it goes back to sleep.

The code that implements this *boot protocol* is mostly in the files `kernel/arch/armv7/boot.S` and `kernel/arch/armv7/boot_driver.c`, if you're interested.

You don't need to implement this boot protocol yourself, the CPU driver provides it via an invocation on a special capability: the *kernel cap*. This capability is only provided to privileged user-space processes, such as the *monitor* in normal Barrelfish, or your `init` process. This capability lets you do all sorts of wild and wonderful things, but for the moment we only care about one of them: *Coreboot* (in Section 3 we'll use another kernel cap invocation to *forge* RAM capabilities).

Now, using Coreboot isn't free — in keeping with its microkernel heritage, Barrelfish requires you to do most of the heavy lifting in user space, while the CPU driver just executes the truly sensitive operation (writing the boot record and waking the core). You need to prepare the new CPU driver to boot, and provide it with a certain amount of memory, and information. Specifically, you need to do the following:

- *Allocate memory*. The new CPU driver needs the following:
  - Memory to load its *relocatable segment* (see the next step).
  - A `core_data` structure (`struct arm_core_data`).
  - Space to load the `init` process. This should be at least `ARM_CORE_DATA_PAGES`×`BASE_PAGE_SIZE` bytes.
  - A *URPC* frame, that you'll use to pass initialisation parameters in Section 3.
  - A new *kernel control block* (KCB), which is the root structure holding the state for a kernel instance (CPU driver).

  Except for the KCB, you can allocate these as separate frames, or in one contiguous block. To obtain a new KCB, you can simply retype a RAM cap, as for other objects. To initialise your `core_data` structure, you should *clone* the running kernel (see `invoke_kcb_clone` in `include/aos/kernel_cap_invocations.h`). This will give you a partially-filled `core_data` struct, in which you just need to fill the fields for the memory you have just allocated, and the initial module you'd like to load.

- *Load and relocate the CPU driver*. Finding and loading the ELF file for the CPU driver is exactly the same as for the user-level processes you've already spawned.

  Loading the image is a little different however: All CPU drivers cloned from the same ELF share a text segment. This means that you only need to load and relocate a subset of the data segment, and the global offset table (GOT). We have provided a support function, `load_cpu_relocatable_segment` in `lib/aos/coreboot.c`, which will handle the loading and relocation for you. Remember that the 'virtual' addresses referenced there are *kernel-virtual* i.e. relocated inside the kernel window. You can use `frame_identify` to find the kernel-virtual address of a frame.

- *Fill in the `core_data` structure*. You need to fill in the appropriate fields, so that the newly-booted CPU driver knows where to find the initial task (`monitor_module`), the memory for the init task (`memory_base_start`), and the kernel command line.

- *Clean the cache*. As discussed in the lecture, you need to ensure that the data you've just written is visible to an uncached observer (the core will boot with the MMU off). You'll need at least a barrier, an invalidate and a clean.

- *Invoke the kernel cap*. Use `invoke_monitor_spawn_core` to boot core 1. If you've done everything right, you should see messages from the booting kernel (`"kernel.01 ..."`).

Note that the Barrelfish codebase distinguishes between the BSP (bootstrap) processor and APP (application) processors. This distinction and naming originates from Intel's x86 where the BIOS chooses a distinguished BSP processor at startup and the OS programmer is responsible for starting the rest of the processors (the APP processors). Although it works a bit different on ARM, the names still fit well enough.

See the lecture slides for more information on Coreboot.

3 Multicore Memory Management

Before you can actually start applications on the second core, we will have to sort out another design question: **How do you manage your available memory between two cores?**

Currently, Barrelfish's design is based a single memory management service which manages all physical memory on behalf of all the cores, and handles requests from all applications. This approach relies on the ability to *communicate across cores* and the ability to *find the process that is responsible for memory management*.

We recommend that you simplify this problem by splitting the memory between the cores, and let one application on each core provide a memory management service for applications on that core. This way, you can re-use most of your self-paging code on both cores.

There are many ways to tell the second core which part of memory it is responsible for. For the moment, you can just place a few words of data in the shared URPC frame that you allocated in Section 2. In the next milestone, you'll reuse this frame to implement a proper UMP-style communication scheme, as discussed in lectures.

Given some way to tell the second core which address ranges it should use, what should you pass to it? Look at `usr/init/mem_alloc.c` in the supplied code. This function initialises the RAM allocator by taking regions from the `bootinfo` structure, which is passed to init on the BSP core by the CPU driver, describing the memory layout. Each region in this list is defined by an `mmap` line in `hake/menu.lst.armv7_omap44xx`. There are several ways you could break memory up: split the RAM caps, or modify the memory map to consist of several smaller chunks. They're all good.

In order to actually use one of these address ranges on core 1, you'll need a cap to it. But where will that cap come from? Recall that Barrelfish CPU drivers have completely partitioned state — you cannot copy a capability from a CNode managed by one CPU driver, into a CNode managed by another. So how are capabilities transferred between cores in Barrelfish? In a normal Barrelfish system, this is done through the *monitor*, a privileged user-level process, that acts as an extension of the trusted kernel, which is able to do things that the CPU driver can't, like block, or communicate over shared memory channels.

We already saw one of the monitor's special powers: booting cores. The second one we need to use now is its ability to create, or *forge* capabilities. Have a look at `include/aos/kernel_cap_invocations.h`. You'll see here both the KCB clone and spawn core invocations we've already used, alongside the `create_cap` invocation that you can use to create capabilities out of thin air. We've wrapped this (**extremely dangerous**) invocation for you, in `lib/aos/capabilities.c`, into two calls: `ram_forge` and `frame_forge`. These allow you to create RAM and Frame capabilities, given only a base address and a size. You must *only* use these operations to transfer resources between cores — as you don't have a full monitor to keep track of them, you can't safely invoke these capabilities on two cores at once, or delete them. All other capabilities should be derived from the root RAM caps that you create here, and can be used exactly as normal — only the forged caps are special.

You now have all the mechanisms you need to transfer RAM to core 1, and bootstrap allocations there.

The last thing you need to do is to give your init process on core 1 access to the `bootinfo` struct. Note that the capabilities to this and all ELF modules are available on core 0, in a special CNode (see `include/barrelfish_kpi/init.h`). You need to come up with a way to make this data available one core 1, using some combination of the above mechanisms.

By end of this step, you should essentially have two instances of a self-paging system running on two different cores!

4 Lab demonstration

You are expected to demonstrate the following functionality during the lab-session:

- Show that the second core is up
- Applications on each core are able to handle pagefaults
- You can run applications on the second core

Once your code is demonstrated, you should submit it over Online submission system (accessible only from ETHZ network) before midnight on Friday night / Saturday morning.