ADVANCED OPERATING SYSTEMS
Milestone 7: Distributed Capability Operations
Fall Term 2016

Assigned on: **25.11.2016** Due by: **16.12.2016**

1 Overview

In this milestone you will bring together the bits and pieces you've implemented so far and make the last steps towards a complete operating system. In this project the goal is to implement the missing pieces which are necessary for the capability system to work as a single entity across both cores of the Pandaboard.

This milestone is the last implementation milestone. You are given a lot of design freedom, but you will have to be able to explain and defend your design choices.

The project is structured in such a way that there are logical steps that build on each other, and will have a lot of opportunities where you can implement functionality that is not necessary for the project work to be given a passing grade but which will certainly improve your grade assuming that all the core functionality outlined in the project goals is working correctly.

The required work consists of:

- Implement transferring capabilities between the init processes on both cores.
- Forward `libaos` capability operations that need synchronization to init on the same core.
- Implement `cap_revoke` for a single core.
- Implement `cap_delete` for capabilities for which copies exist on both cores.
- Implement `cap_revoke` for capabilities for which copies exist on both cores.
- Implement `cap_retype` for capabilities for which copies exist on both cores.

Further goals:

- Allow the init domain to use standard `cap_retype`, `cap_delete`, and `cap_revoke` on capabilities which require synchronization.

- Change your system to use your capability transfer functionality to provide a large RAM region and the bootinfo structures to init on core 1.
- Send a capability from an arbitrary domain on core 0 to an arbitrary domain on core 1.

2 Background: Distributed capabilities

The CPU driver you have been using for the duration of the course already has a lot of the lowest level of functionality that is needed for Barrelfish's distributed capability design. There are a fair amount of invocations that deal with different steps for these distributed capability operations. You will find convenience wrappers for those invocations in `usr/init/distops/invocations.{c,h}`.

The design utilized in Barrelfish, which is what these invocations were designed for, has the following properties:

a) Each capability has an owner, which is one core in the system.

b) Each capability in each set of capabilities that refer to the same object ("copies") has the same owner.

c) The owning core for a capability c must always have at least one copy of c.

If you wish, you can use these three properties as guidelines for your own design.

Given those three properties, things you need to think about are how to manage ownership in the case where you delete the last copy on the owning core, and whether it might make sense to voluntarily move ownership between cores. Terminology you will see in the provided code is that capability copies on the owning core are sometimes called "local", which capability copies on non-owning cores are sometimes called "foreign".

If you want to invent a different design yourself, you should be aware that you might have to modify the CPU driver for your system to be able to implement all the required operations. Additionally, a lot of the information presented below will not necessary apply for your design, and you will have to make sure that "normal" capability invocations (cf. `lib/aos/capabilities.c` and `kernel/syscall.c`) will correctly signal user space whenever an operation needs synchronization (e.g. `SYS_ERR_RETRY_THROUGH_MONITOR`.

3 Getting prepared

For this milestone, you will need to pull the new changes from the repository. This will add a number of new invocations and other supporting code to init. (`/usr/init/distops`).

Run `Hake` again and do a clean build:

commands to run

```
make rehake
make clean
make PandaboardES
```

4 Transferring capabilities between the init processes on both cores

A good place to start your implementation of a distributed capability system is to actually be able to correctly send capabilities from init on one core to init on the other core. This involves making sure that each capability's owner information is set to the core which will be responsible for synchronizing complicated operations on that capability. In the provided code there are a number of functions that will be useful to manage ownership and remote relations of capabilities:

Ownership and remote relation management methods

```
errval_t monitor_get_cap_owner(struct capref croot, capaddr_t cptr,
                               int level, coreid_t *ret_owner);
errval_t monitor_set_cap_owner(struct capref croot, capaddr_t cptr,
                               int level, coreid_t owner);
errval_t monitor_remote_relations(struct capref cap,
                                  uint8_t relations, uint8_t mask,
                                  uint8_t *ret_relations)
```

As you can see, these functions have "monitor" in their function names, hinting at the fact that in Barrelfish they're used in the *monitor* which is the domain running on each core in the system and executing privileged operations with unbounded execution time, so most of the CPU driver operations can be completed in bounded time, allowing the CPU driver code to run with interrupts disabled.

What you actually send to the other core is the actual `struct capability` of the capability that you are transferring. In order to create a correct capability on the receiving core you will use `monitor_create_cap()`, which is similar to the function you are using to create a RAM capability for the memory server on core 1.

Given the design properties mentioned earlier, you can see that you will need to properly set the owner of each capability to either core 0 or core 1. The remote relations for each capability indicate whether a capability has copies, ancestors, and descendants on the other core.

Helper functionality and bit definitions for the remote relations property of the capabilities is provided to you in the file `include/barrelfish_kpi/distcaps.h`.

Assuming you implement this capability transfer correctly, you will get the error `SYS_ERR_RETRY_THROUGH_MONTIOR`, not only for revokes but also for `cap_retype` and `cap_delete` if those functions are called on a capability for which copies exist on both cores.

EXTRA CHALLENGE

4.1 Ownership transfer

Once you can transfer capabilities between the two cores you may want to implement a variant of the capability transfer which makes the receiving core the new owner of the transferred capability.

5 Forward `libaos` capability operations that need synchronization to init on the same core

To handle these operations that return `RETRY_THROUGH_MONITOR`, you will have to implement a set of RPCs to init which will act as the synchronization point on each core for capability operations that need synchronization.

This can be done quite easily by providing init access to your domain's cspace. This can be accomplished by an RPC call that sends the domain's root cnode (`cap_root`) to init and further provides capability address of the capability on which the operation was originally invoked as an argument.

In this step it is sufficient for init to print out some information about the capability which needs to be retyped/deleted/revoked. From init, you can easily refer to capabilities that exist in other cspaces by creating a `domcapref` from the information received through the RPC (cf. `usr/init/distops/domcapref.h`). Init also has the means to inspect capabilities (cf. `monitor_cap_identify()` and `monitor_domains_cap_identify()`), which you can use to acquire a copy of the actual `struct capability` from the CPU driver. To produce a human-readable format from that information there is a function `debug_print_cap()` in `libaos`.

6 Deleting CNodes and Dispatchers

For now, we are taking a step back and are looking at the case where we delete the last copy of a capability that itself contains references to other capabilities. This capability is most likely a CNode, but the Dispatcher control block capability also contains a copy of the dispatcher's root CNode.

Assuming that the capability is a CNode, and the copy we are deleting is the last copy, we need to make sure that all the capabilities that are contained in that CNode are deleted as well.

As we cannot reliably predict the execution time for such an operation, we split this operation into multiple steps. The first step is iterating through the CNode and marking all the capabilities it contains for deletion. This step is actually done by the CPU driver when init calls `monitor_delete_last()`. After that first step there is one step for each capability that got marked for deletion. These steps can be executed by repeatedly calling `monitor_delete_step` and `monitor_clear_step`. This will process capabilities that are marked for "delete" and "clear" respectively. Clearing a capability is the step where the CPU driver will create a new capability from scratch if there is no capability referring to the underlying object left in the system.

6.1 Delete Stepping Framework

For your project, we provide a set of functionality that wraps the delete and clear steps in a easy-to-use event-based framework, which you can find in `usr/init/distops/deletestep.c`. In order to use the framework you need to initialize it by calling `delete_steps_init(struct waitset *ws)` with a waitset on which you are doing `event_dispatch()` as the argument (e.g. the waitset you are using to process events on LMP channels).

If the framework is initialized, processing delete and clear steps is a matter of setting up a callback which will be called when the delete and clear queues are empty by using `delete_queue_wait()`, and making sure that queue processing is paused and resumed at appropriate points using `delete_steps_pause()` and `delete_steps_resume()` (processing delete steps while we are inserting new elements might be a bad idea).

7 Revoking capabilities

When revoking a capability there is always the possibility of suddenly having more work of our hands. Consider the case where we revoke a RAM capability that was retyped to a CNode. Now all the issues discussed in the previous section apply to that revoke, as we need to make sure that before we finally delete a CNode all its contents are properly deleted.

For revoke we again split the work in multiple steps. The first step is marking all copies and descendants for deletion with `monitor_revoke_mark_target`. After that we need to process the delete and clear queues the same way we did for delete.

8 Deleting capabilities that exist on both cores

As you can imagine, deleting a capability for which copies exist on both cores needs synchronization. One reason we need synchronization is that what we discussed in section 6. A further complication is rule #3 in our example design “The owning core of a capability c must always have at least one local copy of c”. This means that deleting the last copy of a capability on its owning core in the presence of copies on the other core needs to transfer ownership to the other core (or delete all copies on the other core, if ownership for the capability’s type is not transferable).

Exactly how you do this synchronization is up to you. Barrelfish’s implementation, which needs to worry about systems with more than two cores does all operations that require synchronization as a two-phase commit (2PC).

2PC matches nicely with the mark & sweep scheme discussed for deleting and revoking capabilities in the previous sections. The mark phase is the first phase of the two-phase commit and we trigger the sweeping when we send or receive the commit of the 2PC protocol.

This protocol needs to happen between the init processes and to implement the protocol you can send messages and/or RPCs over the UMP channel between the two init processes.

9 Revoking capabilities that exist on both cores

As you can imagine, doing a capability revoke – which is essentially a delete of all copies and descendants of the target capability – in a system where copies and descendants of the target can can exist on multiple cores needs to be synchronized for the same reasons an individual delete needs to be synchronized.

We can also use the same 2PC synchronization where we mark first and then sweep when we get the commit.

10 Retype capabilities that exist on both cores

When we try to retype a capability for which copies exist on both cores it is necessary to check that the requested retype doesn't violate the guarantees the capability system makes about types and never allowing capabilities to refer to overlapping regions. The only case where we allow overlap is for parent capabilities, which must completely contain the regions referred to by their children.

In order to check this for this overlap, we need to forward the retype request to the other core which then needs to check that it does not hold any capability that would produce a retype conflict. To do this check you can use `monitor_is_retypeable()`, which will execute all the checks that are done during a `cap_retype()` invocation, but will not actually produce the retype result.

EXTRA CHALLENGE

11 Use standard capability operations from init

As you may have noticed, `init` itself does not work when using a `libaos` capability function that causes an RPC. A general solution to this problem – using the same library code in the domain handling the RPC as in all other domains – is to provide a special RPC backend which just locally calls the right handlers. Implement such a "local" or "loopback" RPC backend, so init can use standard `cap_revoke` and deletes and retypes for capabilities with copies on both cores.

EXTRA CHALLENGE

12 Use "real" capability transfer when inititalizing init on core 1

Now that you have a completely working distributed capability system, you can use your cross-core cap transfer to send the initial RAM cap and the bootinfo capabilities to init on core 1.

EXTRA CHALLENGE

13 Send capabilities between any two domains

Now that you have a completely functional distributed capability system, you can also look at sending capabilities from a domain on one core to a domain on the other core. This must involve init on both cores as you cannot transfer capabilities between cores without having a trusted domain to create the capability on the destination core. It is perfectly acceptable that any message between domains on different cores which contains a capability is routed via the init domains, even if you have direct UMP channels between the domains. However, this decision should be made transparent to the client code.

14 Lab demonstration

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

- You can send capabilities between the two init processes
- You can demonstrate init receiving and processing capability operations
- You can correctly handle `cap_delete` for any capability
- You can correctly handle `cap_revoke` for any capability
- You can correctly handle `cap_retype` for any capability

Once your code is demonstrated, you should submit it over Online submission system (accessible only from ETHZ network) on Friday 16.12.2016 midnight (23:59h).