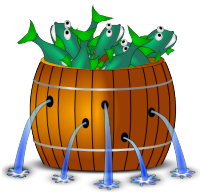

ADVANCED OPERATING SYSTEMS
Milestone 6: Fast User-Level Message Passing and RPC
Fall Term 2016

Assigned on: **11.11.2016** Due by: **25.11.2016**

1 Overview

In this milestone, you will implement a full inter-core messaging and RPC system, to replace the rudimentary shared frame you used in milestone 5. Using this, you will extend your RPC system to allow processes to be spawned and managed on core 1, from core 0 (and vice-versa).

There's no additional code to check out for this milestone, and we have deliberately not specified how you are to implement your RPC system — by this point, you'll be building on your own earlier work, and taking charge of the design!

2 Getting prepared

To prepare for this assignment you must modify the `process_spawn` RPC call to include the ID of the core on which you want to spawn the new process:

```
errval_t aos_rpc_process_spawn(struct aos_rpc *chan, char *name,
                               coreid_t core, domainid_t *newpid);
```

This is the RPC signature that we will test against.

It will also be helpful to have working user-level threads in order to make some of the implementation for this milestone easier (e.g. polling on the cross-core channel).

3 Inter-Core Communication

Having what essentially amounts to two single-core systems that happen to share a multicore platform is not that interesting. It also implies that you need to run two shells (or equivalent) on two cores to start other processes on those cores. While there might be a use-case for such 'co-located' single-core systems, we want to have a proper multicore system where the different cores can share functionality such as the user interface.

Needless to say, implementing a shared user interface (and a proper multicore system in general) requires communication channels between applications on different cores. So in this step, you will implement a shared-memory based communication channel between the cores.

3.1 Sharing a Frame Between Cores

As Barrelfish draws on the micro-kernel design philosophy, we prefer to push most functionality out of the kernel. This also means that we try and provide a form of inter-core communication which can be done directly from user-space without involving the kernel in every message exchange.

A simple form of user-space communication (not even necessarily across different cores) is to share a region of memory between the applications that want to exchange information and then read and write that region using an established protocol.

You must implement such a simple shared-memory based, user-mode communication channel. There are multiple ways of implementing such a channel. The following is one, but feel free to experiment with other designs. See also the description of UMP in this week's lecture.

To begin with, you'll need some shared memory. Recall the URPC frame that you used in milestone 5, to pass RAM parameters to the second core. URPC stands for user-level RPC, and this frame is normally used in Barrelfish to establish an initial communication channel with the monitor process on a newly-booted core. We don't have a monitor, so you're free to reuse this frame for your own protocol. On core 1, the URPC frame is available in the task CNode. Make sure that you map this frame cacheable — using an uncached mapping is fine for testing, but it's cheating in the final submission (it's *really* slow).

Also, remember what we covered in lectures regarding ARM's weak memory model. You can't count on your stores being observed by the other core in the same order that you write them down! You will need to insert barriers as appropriate to ensure that your protocol is correct. You will need to explain your implementation to your tutor, and demonstrate that you understand why it is correct.

3.2 Establishing a communication protocol

As we now have shared memory between the two init processes, we should establish a protocol which enables meaningful communication between them. As our requirements from this communication channel are modest, a simple protocol should suffice.

We will run only one shell which will be responsible for spawning applications on both cores. Therefore, we only need master-slave communications. As the master application can wait while a process is starting on the remote core, we can implement this communication as a remote procedure call (RPC).

Essentially, we need to support a *remote spawn* message as well as a corresponding response on this channel. In this remote spawn message, `init.0` should send the name of an application to `init.1` which should then start this application and report the status (success/failure) to `init.0`. The shell on the BSP core can send requests for *remote spawn* over LMP to `init.0` which then forwards the message to `init.1` (using our new shared-memory communication channel) for actual execution. Similarly, responses can go back from `init.1` to the shell on the BSP core.

To make this work with your existing RPC interface — which didn't consider the fact that you might want to spawn an application on another core — you have to extend your existing `aos_rpc_process_spawn` RPC to have the following signature

```
errval_t aos_rpc_process_spawn(struct aos_rpc *chan, char *name,
                               coreid_t core, domainid_t *newpid);
```

and take the necessary steps inside your process management system to enable spawning processes on the second core over a channel like the one described above.

Once you're able to send messages between cores, you're ready to start launching processes. Remember to make sure that you've correctly mapped the `bootinfo` structure, where the ELF images live, into your address space on core 1.

You should now extend your URPC protocol, to allow all RPC operations to be fowarded between cores. A sufficient implementation is to route all RPC calls via the init process (or equivalent), which forwards them on behalf of user-level applications.

You must be able to reach any RPC server on core 0, from an application on core 1, and vice versa. You should provide an interface for applications to *bind* to servers, and then use this binding to make RPCs.

Note that you need to make sure that you don't accidentally forward messages that should be handled by an application on the same core.

EXTRA CHALLENGE
Implement more generic communication between processes across cores so that any two processes can setup a direct channel with each other. This will require a more sophisticated binding operation, as the processes will need to share memory.

EXTRA CHALLENGE
Currently applications need to be told (by the application starting them or the programmer) how they can reach system services. This has obvious scalability problems when the number of system services grows. A classic solution to this is to implement yet another system service that knows what other services exists (a "nameservice"). Obviously applications need to be told how they can reach the nameservice but they can lookup any other service using their connection to the nameservice. Maybe your system would benefit from such a name service.

4 Lab demonstration

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

- You can start applications that run on the second core from the BSP core.
- You can manage processes across cores (list processes, etc).
- Processes on the second core can access RPC services on the first core e.g. memory server, or serial console.
- Present *and explain* your user-level messaging protocol.

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.