1 Overview

By now, you should have a kernel and a user-space processes. You can also refer to physical memory using capabilities, and you should be able to use upcalls to service page faults, assuming you still have enough physical memory available.

To get any further, you need to be able to communicate between processes, and this is the theme of this week’s assignment. Once you have inter-process communication, you can write servers to manage resources (such as memory), and acquire these resources to create new processes dynamically.

This is quite an involved milestone, and so we’ll give you two weeks for it. Thus there is no grading this Friday (unless you are still behind with last week’s milestone), but there will be a consultation session if you need to check in with the assistants about your progress.

The work consists of:

- Simple message passing between two initial domains
- Passing capabilities between the domains
- Simplifying domain initialization
- Creating and using a memory server

2 Getting prepared

For this milestone you will give you patch fixing few minor issues in the code. You can apply the patch with following command:

    patch -p1 < /pub/aos/handout/m3.patch

You also need to adjust the milestone variable in your build directory’s hake/Config.hs, changing the variable PANDABOARD_MODULES in symbolic_targets.mk and changing the file menu.lst.pandaboard as shown below as well as running the following commands in the build directory:
symbolic_targets.mk

PANDABOARD_MODULES=\n   armv7/sbin/cpu_omap44xx \n   armv7/sbin/init \n   armv7/sbin/memeater

menu.lst.pandaboard

kernel  /armv7/sbin/cpu_omap44xx loglevel=4
module /armv7/sbin/cpu_omap44xx
module /armv7/sbin/init
module /armv7/sbin/memeater

commands to run

make rehake
make clean

3 Simple local message passing

The next step is to pass small messages between processes. The Barrelish CPU driver provides a simple mechanism to do this, based on *endpoint capabilities*. The system is called “LMP”, for “Local Message Passing”, and is based on LRPC and similar facilities in the L4 microkernel.

First, the theory: you can create an endpoint capability by retyping the capability for the domain control block. There is a function to do this in the CPU driver, and a corresponding system call to do this in user space.

However after retyping the domain control block capability to an endpoint, you are not quite finished. Barrelish doesn’t let you use that endpoint for sending messages because each endpoint needs to have a buffer associated with it.

You can create an endpoint with an associated buffer by *minting* the capability you created before into another slot and by giving the mint operations the proper arguments for the buffer offset and size (we have \#defined these values for init’s initial endpoint — FIRSTEP_OFFSET and FIRSTEP_BUFLEN).

Note: the value for FIRSTEP_OFFSET has been determined experimentally and as long as it satisfies the following formula all that matters is that you use the same value everywhere:

\[ \text{FIRSTEP\_OFFSET} \geq \text{get\_dispatcher\_size()} + \text{offsetof(struct\_lmp\_endpoint, k)} \]

Once you’ve got a proper endpoint capability in your own domain, you can think of it as like a non-blocking server socket: when a message arrives for the endpoint, an upcall is made to your domain with additional information indicating which endpoint the message is for.

Conversely, to send a message to a domain you need to have acquired a copy of this endpoint capability. This can then be “invoked” (again, there is a system call for this) which sends a message containing a small set of arguments in registers.

Like all networking systems, there is a bootstrapping problem: how does one domain get hold of an endpoint capability for another domain? To start with, we’ll fix this by installing the right capabilities in each domain when they start.

You should modify the code in the CPU driver which creates the two domains to create their own endpoint capabilities and store these in their CSpaces, in well-known slot locations. Furthermore, put a copy of init’s endpoint in the other domain’s CSpace, also in a well-known place.

Now, you should be able, at a very low level, to send a message from the second domain to init by invoking, in the second domain, the endpoint capability which is a copy of the one retyped from the
The best way to demonstrate this is to instrument the upcall handler/thread scheduler in each domain.

The best way to understand both the API for sending a message, and the in-kernel implementation that performs the message send, is to look at the function `handle_invoke` in `kernel/arch/armv7/syscall.c`. You’ll see that a number of flags can be specified when sending a message, which give hints as to what domain to run next. The rest of the code to implement LMP is in `kernel/dispatch.c`.

Note that sending a message will not automatically succeed – for example, if the receiving domain is not a position to process the message, you will get a negative acknowledgement back from the system call. In this case, the best approach is to yield the processor to allow the other domain to run, and try again later.

Once you’ve got this working, you can modify the thread scheduler to allow a thread to block on a specific end-point capability waiting for messages for it. When the upcall occurs, the thread scheduler should identify the endpoint, work out if a thread is blocked on the endpoint, and if so, unblock it and allow it to read the message.

**EXTRA CHALLENGE**

Traditionally, operating systems (in particular microkernels) have tried to make communication between two processes on the same core as fast as possible. You can measure how long it takes between sending a message and receiving it on the other side using the performance cycle counter on the ARM (it’s on CP15).

Our implementation in Barrelfish isn’t bad, but it’s almost certainly not as fast as it could be. See if you can make it faster (and demonstrate this), without removing any of the functionality it currently offers.

### 4 Initialize user-level LMP machinery for kernel-created endpoints

In a next step — as you generally would like to have a nicer interface to the message passing facilities than having to marshal messages into system call arguments by hand, you’ll have to initialize the user-level LMP system for the endpoints you created in the kernel.

This involves setting up `struct lmp_chan` structures for both ends of the connection. You’ll have to make sure that on init’s end, the local endpoint corresponds to the endpoint we’ve been using for the previous step. Here you can use the function `lmp_endpoint_setup` which will allow you to specify which region of the dispatcher the endpoint (which we’ve already created in the kernel) will use and the function will setup the user-level structure accordingly. Additionally you will have to set the `local_cap` field of the LMP channel to the endpoint capability.

As you’re now going to be using Barrelfish’s event handling along with the local message passing described above, here’s a short introduction to the event handling mechanism: at the core of Barrelfish’s event handling lies the `waitset`, which forms the connection between a `thread` and an `event`. Here, an event is represented as a closure (function and argument pointers). For us, the most important bit is that these events are raised by message channels (e.g. an LMP channel) in response to activity (such as being able to receive a message).

Coming back to waitsets, as activity on message channels leads to them raising events, each channel has to have a waitset associated with it’s receive handler and there has to be a thread in the domain `dispatching` events on that waitset (this might amount to simply calling `event_dispatch` on the waitset in an infinite loop).

In order for the channel to know what event to raise in the case of a message being ready to be received there’s a function called `lmp_chan_register_recv` which you can use to register an event (remember: an event is simply a function pointer with an associated argument, i.e. a closure). What you probably at least want to do inside that function is to call `lmp_chan_recv` to actually get the message and at the end reregister an event (using `lmp_chan_register_recv` again) so that if there’s more messages you can...
receive those as well.

On the sending side that has a `struct lmp_chan` for the channel, you can simply call `lmp_chan_send()` (or one of its variants, in `include/arch/arm/barrelfish/lmp-chan-arch.h`) with the appropriate arguments.

## 5 Passing a capability over LMP

Basic LMP sends a small number of data words, using registers. However, you also need to be able to pass capabilities. This amounts to creating a copy of a capability held by the sender in the CSpace of the receiver.

For this to happen, the receiver has to say in advance where in its CSpace the incoming capability is to be stored.

The next step is to use this facility to pass a capability in a message, along with other values.

There’s functionality in the user-level LMP machinery to set an empty slot for receiving a capability.

First, demonstrate creating a new endpoint in one domain, passing it to the other, and have the receiving domain then call this new endpoint back.

You’ll probably realize that what’s going on here is basically a form of RPC (Remote Procedure Call). As with RPC, you’ll need to settle on a convention (actually, part of the protocol) for specifying what each message means, beyond simple the endpoint that it’s called on.

### EXTRA CHALLENGE

LMP on Barrelfish with ARMv7 processors can only transfer a few 32-bits words and an optional capability. Of course, messages often need to be bigger than this. There are several approaches to this problem.

First, you can write a small piece of software (a “stub”) which breaks a large message into several smaller ones, and another stub on the receiving side which assembles the pieces before delivering the larger message.

Secondly, you can use a special area of memory on the sender and receiver, and have the kernel copy more data between these buffers during the call. This is tricky, because you have to make sure no other thread is using the buffers when a message is sent.

Thirdly, you can create a special area of shared memory between two domains, and make sure both domains have a capability to it. They can then map this into their own address spaces, and use it to pass messages. This is a lot of work to set up (in particular, the exchange of capabilities for the memory) and tear down when done (likewise), but is the best approach when messages are very large, and the only way to go between cores (as we’ll see later in the course).

## 6 Use your LMP abstractions to implement a given RPC interface

Now that you have a usable system for sending and receiving LMP messages, you should implement the following RPC interface on top of your LMP abstraction (or directly on top of an LMP channel). The interface has only a few functions for this milestone and we’ll expand it for new functionality in later milestones.

```c
struct aos_rpc {
};
```
errval_t aos_rpc_send_string(struct aos_rpc *chan, const char *string);

errval_t aos_rpc_get_ram_cap(struct aos_rpc *chan, size_t request_bits,
                               struct capref *retcap, size_t *ret_bits);

errval_t aos_rpc_init(struct aos_rpc *rpc);

As you can see from the functions you’ll have to implement two RPCs for this milestone, an RPC that sends a string, and an RPC that requests a RAM capability.

The idea behind this RPC interface is that often operations like requesting a RAM capability from some other domain (most likely a memory server c.f. the next step) are actually comprised of sending a message and waiting for the reply to that message while still processing other unrelated messages that arrive over the channel.

The purpose of the struct aos_rpc is to keep state for your RPC channel (e.g. the underlying LMP channel, the currently pending replies). You may change the function signature of aos_rpc_init() to accommodate setting up an RPC channel’s state with whatever your implementation needs.

Note: If you change the function signatures of the actual RPC calls in the interface you’ll need to give a good explanation why this is necessary and document the changes as we will write test programs using that interface.

7 A memory server

You have now got all the basic features required for inter-process communication: bootstrapping communication, passing data, and passing references to further communication endpoints.

The next step is to do something useful with this: implement a memory server which can allocate regions of physical memory to other domains, and hand over capabilities to those domains.

You can implement the server in any way you like. One simple way is to run it in the init domain, and have other domains use their existing communication channels to init to request and receive memory.

More elegant, but more work, is to implement the memory server as a separate domain (BarreLFish itself employs a variation of this technique). This also requires you to provide a way for domains to request an endpoint capability for the memory server from some other domain which they can already talk to (such as init); essentially, this involves implementing a simple name server.

You should also think about the operations that the server supports. For example, clearly you need to make sure that each client receives capabilities to disjoint areas of memory in response to their requests. You might also want to limit the quantity of memory each client receives, to prevent a client from grabbing all the available physical memory.

As a final step, allow new domains to perform self-paging and memory allocation on demand using your new memory server and the get_ram_cap RPC call.

8 Lab demonstration

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

- That you can correctly invoke an endpoint capability and receive a message on the same capability in another domain.
- Bootstrap new domains using only an init endpoint capability as their startup information.
• Demonstrate that your RPC implementations are working.

• Demonstrate and explain a memory server process allocating RAM caps among multiple domains.

Try to make sure your code is bug-free. We’ll expect you to demonstrate a system that can handle an arbitrary amount of domains talking to each other and each of those domains should be able to talk to the memory server when it needs more physical memory.

Submission

Don’t forget to submit your code as a tar-ball through the submission system accessible from course website before the specified deadline.