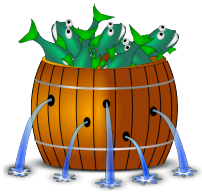

CS378: MULTICORE OPERATING SYSTEMS IMPLEMENTATION
Milestone 3: Message passing and RPC interface
Spring Semester 2017

Assigned on: **27.02.2017** Due by: **20.03.2017**

1 Overview

By now, your operating system should be able to manage physical memory and spawn new user-space processes. To get any further, you need to be able to communicate between processes, and this is the theme of this week's assignment.

The particular flavour of inter-process communication that you will be working towards for the rest of the milestones is RPC, that is remote procedure calls. You can think of RPC as a procedure call that happens to call into another process. RPCs provide a uniform interface on which you can implement server-client communication with different system services that provide functionality such as physical memory management.

In this weeks assignment, you will implement the core RPC mechanisms for both client and server side, and you will implement and use the first services: the memory server, and serial output over RPCs to init.

Please note that you will have to implement more RPCs and servers in future milestones, so do take some time to think about a suitable design which allows you to easily implement new RPCs and new RPC servers in the future.

The work consists of:

- Simple message passing between two initial domains
- Passing capabilities between the domains
- Using a memory server in the init process
- Using `init` as serial driver.

2 Getting prepared

For this milestone, you will need to pull the new changes from the repository. This will add a new domain (`/usr/memeater`) to your source tree and to the list of modules to be built. Make sure `memeater` is also added to your menu.lst. Note: have a look at `memeater`'s Hakefile. You will see there is no option `-e _start_init`. This is required for init, because init has a slightly different initialization path that stops before the message passing infrastructure is initialized.

Run `Hake` again and do a clean build:

commands to run

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

3 Background: Simple local message passing

The next step is to pass small messages between processes. The Barrelfish 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 from the dispatcher capability. 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 dispatcher capability to an endpoint, you are not quite finished. Barrelfish doesn't let you use that endpoint for sending messages because each endpoint needs to have a buffer associated with it. You can use the *mint* operation for this.

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 the receiver's 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 are getting spawned. You should modify the spawning code to pass a newly created endpoint capability at a well-known location in the new domain's CSPACE.

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 `init`'s DCB. Make sure that `init` also has the corresponding endpoint capability of the other 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 in 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.

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). (2 bonus points)

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. (up to 5 bonus points, depending on how much you do here)

4 Implementing your message passing abstraction

Dealing with the raw, syscall interface of LMP is a bit tedious, so you want to provide appropriate abstractions. In the end, you will need to implement the specified RPC interface. We will provide you with the low-level abstractions such as `lmp_endpoint` and `lmp_chan`. Your task will be to provide functionality to distinguish different message types, marshalling unmarshalling of messages and RPC semantics.

4.1 LMP Endpoints and Channels

We already provide the code to manage raw LMP channels in `lib/aos`. In both the parent and child process you can use the function `lmp_chan_accept` to create and setup a new LMP channel. `lmp_chan_accept` creates a local endpoint by calling the function `lmp_endpoint_create_in_slot`, and will set the channel's remote endpoint to the one passed as an argument.

In order to properly setup a communication channel between parent and child, the parent needs to create the its end of the LMP channel when spawning the child, and pass its local endpoint to the child by putting it in a well known location in the child's cspace during process creation. The child can just pass its local endpoint back to the parent by sending it over the LMP channel which it creates using the endpoint provided by the parent as the remote endpoint.

4.2 Sending and Receiving Messages and Capabilities

As soon as you have set up your LMP channels, you can start sending or receiving messages using `lmp_chan_recv` and `lmp_chan_send`.

On the receiving side, you will get a pointer to a message struct containing your payload. Note, you can also receive capabilities, but you will need to provide an empty slot in your CSPACE in order to receive them. Hence, you need to allocate a slot in advance. There's functionality in the user-level LMP machinery to set an empty slot for receiving a capability.

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/aos/lmp_chan_arch.h`) with the appropriate arguments. On ARMv7, you can send up to 9 raw words in one message. Your channel abstraction will need to take care of marshalling the payload into those 9 words.

Note that both, send and receive, can fail either because there is no message to be received or you cannot send because the other side does not receive the messages. One approach would be to busy-wait until you can send/receive. Next we will see what you can do to avoid this.

4.3 Event handling

At this point, you have set up the LMP channel. However, there are still some steps to do until you can use the channel. This includes handling events. Events together with message passing are a central element in Barrelfish. Services wait until they receive a message, then perform certain actions related to the request and – depending on the request – send a reply back.

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). Hence, you may need to provide a waitset when you setup your channel abstraction. You can always use just one waitset (see `get_default_waitset()`) or maybe your implementation of the RPC requires to have separate waitsets.

You can register for receive events on a LMP channel using `lmp_chan_register_recv` (same applies for sending). You will need to provide the LMP channel, the waitset you want to associate the LMP channel with and an event closure to be called when there are new events on the LMP channel. The event closure will be your receive handler and some argument.

4.4 Your channel abstractions

In the end, you have to implement the RPC interface stated below. There are multiple ways to do this, e.g. you can implement the RPC interface on top of the low-level LMP channels or on top of another intermediate abstraction.

In both cases, you need to think of which state you need to track. You can store the channel to init at a well-known location. Furthermore, you will need to define a protocol and message format, as you will need to be able to handle multiple message types. In particular, you will need to implement functionality to send numbers, request new RAM capabilities and also handle strings. You can limit the maximum size of the string or support sending variable sized strings (see extra challenge below)

You have complete implementation freedom. You will have to conform to the RPC interface (see `include/aos/aos_rpc.h` for the full RPC interface) and document your implementation. The code excerpt below only reproduces the client-side RPC functions for this milestone and you will have to implement the remaining RPCs given in `include/aos/aos_rpc.h` in later milestones. As you can see from the functions you will have to implement three RPCs for this milestone, an RPC that sends a string, one that sends a number, one that requests a RAM capability, and one last RPC which prints a single character to the serial port.

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.

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 in the interface (apart from `aos_rpc_init()` as mentioned) 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.

```
struct aos_rpc {
    /// state for our RPC
};

errval_t aos_rpc_send_number(struct aos_rpc *chan, uintptr_t val);
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 bytes,
                             struct capref *retcap, size_t *ret_bytes);

errval_t aos_rpc_serial_putchar(struct aos_rpc *chan, char c);

errval_t aos_rpc_init(struct aos_rpc *rpc);
```

Once your channels work, make sure that you setup the RPC client to init in `lib/aos/init.c`, in the function `barrelfish_init_onthread()`.

Once you have a fully initialized RPC client for init, you can use the function `set_init_rpc()` to store it in your application's "core state". This allows you to then use the RPC client for init by just getting it by calling `get_init_rpc()`. See `lib/aos/domain.c` to see how these two functions work.

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. (2 bonus points)

Second, 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. (3 bonus points)

Third, 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). (2 bonus points, but it'll also save you time later)

5 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 as you already have the facilities for managing memory in `init`. Other domains can 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 or to pass the endpoint capability at a well known location in the domain's `CSPACE` upon spawn.

You should also think about the operations that the server supports and the invariants. 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 quanitity of memory each client receives, to prevent a client from grabbing all the available physical memory. Important, state the policies you implement in the documentation.

As a final step, allow new domains to perform memory allocation on demand using your new memory server and the `aos_rpc_get_ram_cap` RPC call. After you obtained a new frame capability, you can use your paging infrastructure from the previous milestone to map it and access the memory on it. Note, you will obtain a RAM capability which needs to be retyped. Barrelfish already has wrapper functions that handles this: `frame_alloc`. You will need to reset the RAM allocator to use your RPC. During the initialization of `libaos`, you can reset the RAM allocator by a call to

lib/aos/init.c

```
ram_alloc_set(NULL);
```

and provide the corresponding implementation in

lib/aos/ram_alloc.c

```
static errval_t ram_alloc_remote(struct capref *ret,
                                 size_t size, size_t alignment)
```

6 A terminal driver

Currently you do a syscall (`sys_print`) when you do a `printf`. You can now use your communication framework to use `init` or another domain as a "terminal" driver. For this purpose you will need a way to send characters (or strings) between the domains. You can use your `send_string` RPC or send a single message. In order to use your own terminal read and write functions you need to set the libc function pointers accordingly. Change the lines in `barrelfish_libc_glue_init` to point to your new read/write functions. For this milestone you need to implement the terminal write functionality.

lib/aos/init.c

```
void barrelfish_libc_glue_init(void)
{
    /* optional */
    _libc_terminal_read_func = aos_terminal_read;
    _libc_terminal_write_func = aos_terminal_write;
    /* ... */
}
```

EXTRA CHALLENGE
Implement the terminal read functionality that requests input characters from the terminal service (`init`). You will need to add another RPC to the interface that allows you to request a character from the terminal service. There is one serial port you will use to read from. You will need to think of a way to multiplex this resource such that you the read characters are not split between multiple domains. (2 bonus points)

7 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.
- Demonstrate that your RPC implementations are working.
- Demonstrate that you can spawn new domains upon request.
- Demonstrate and explain a memory server process allocating RAM caps among multiple domains.
- Demonstrate and explain how your terminal service works.

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.