1 Overview

In this milestone, you will bring up the second core, establish communication between them, and run applications on it.

2 Getting prepared

To prepare for this assignment you should modify the “process spawns” RPC call to include the core id on which you want to spawn the new process:

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

Remember that it might beneficial to you 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 Bringing up second core

This step concentrates on bringing up the second core and the ability to run code on it.

Here is a brief overview of how the bootstrapping process for the second core works: it waits for a signal from the BSP core (an interrupt), and when this signal is received, the application core will read an address from a well-defined register and start executing the code from this address.

So, you can give some work to the other core by simply writing the address of a function to the register and sending the signal. Following are few pointers to the documentation to help you to understand the bootstrapping process in more details.

- Section 27.4.4 in the OMAP44xx manual talks about the boot process for application cores
- Pages 1144f in the OMAP44xx manual have the register layout for the registers that are used in the boot process of the second core.

Following is a code snippet from `start_aps.c` (the defines are in the associated .h file) that you may find useful
// for pandaboard
// Register for communication between two cores
#define AUX_CORE_BOOT_0 ((lpaddr_t)0x48281800)
// Register to tell which code to run
#define AUX_CORE_BOOT_1 ((lpaddr_t)0x48281804)

/**
 * Send event to other core {pandaboard specific}
 */
void send_event(void)
{
    __asm__ volatile("SEV");
}

Note that the Barrelfish codebase distinguishes between the BSP (bootstrap) processor and APP (application) processors. This distinction and naming originates from Intels x86 where the BIOS will choose a distinguished BSP processor for you at start-up and the OS programmer is responsible to start the rest of the processors (the APP processors). Although it works a bit different on ARM, the naming convention is applicable here as well.

3.1 Notes

• Remember that second core will start working with the MMU disabled, so it only understands physical addresses at this time.
• As the second core will be doing function calls of its own, it will need its own stack and some assembly code to setup stack and CPU state. Looking at kernel/arch/omap44xx/boot.S will be a good idea. You may have to provide proper app_core_stack and app_core_init.
• As we just want to get the second core up, you may want to setup and execute a function on the second core in a very early stage of boot-up before things become complicated with MMU, caching and interrupts.
• As soon as both cores start running, you can run into all sorts of concurrency issues. For example, you might want to make sure that only one core will use the UART at given time!
• Make sure that the function executed by the second core never returns!
• You can find out the core-id of given core by calling hal_get_cpu_id.

At the end of this step, you should be able to print something from the application core.

4 Running cpu driver on second core

Now that you can execute a function on the second core, this step tries to run the CPU driver on the second core.

We need a full-fledged CPU driver to make good use of the second core. The code provided by us for milestone 2 contains lots of the functionality to manage the second core already.

4.1 Booting the second core from userspace

Now we come to a design question: Who should trigger booting the second core? Your kernel on the BSP core can start the new kernel on the second core, or it can be started indirectly by an application
Barrelfish’s design delegates the responsibility of booting up other cores to userspace system services for greater flexibility. We provide instructions based on this design here, but feel free to experiment with alternate designs that fit your system.

To boot an application core (i.e., the second core) from userspace, you need a way to send this request to the kernel. You can do that either by implementing a system call or by using a capability invocation on the Kernel capability (which your init domain should have in the task cnode at \texttt{TASKCN\_SLOT\_KERNELCAP}).

There is some code related to starting up another core in the Barrelfish tree you already have. You might want to take a look at \texttt{sys\_boot\_core} and \texttt{sys\_monitor\_spawn\_core}.

Do not worry if you can’t execute the code from the previous step on the second core from user-space using a system call. You will need to do some more work for that.

### 4.2 Starting cpu driver on second core

The next step involves executing a CPU driver on the second core. You can use the same infrastructure as used by the BSP core and start execution at start-address \texttt{start} in the \texttt{boot.S} file. If you do this directly, you will probably end up getting a \texttt{kernel-panic} on one or both cores! The reason behind that is that you are re-using same ELF image, stack, \texttt{got} and global variables on both cores. This means that both kernels are busy modifying each others state whenever they modify any memory location!

So, what you need is another copy of the ELF image for the CPU driver which has been relocated. The steps involved in this are quite similar to the \texttt{spawn\_load\_with\_bootinfo} function in lib/spawndomain/spawn.c. You may want to adapt this function to load the cpu-driver for the second core. You may also take a look at Barrelfish Monitor code but be aware that this version of the code does not work with the Pandaboard and may not work for you out of the box. Also, this code is written for a slightly different design where a trusted domain called the \texttt{monitor} is responsible for starting other cores.

Your code should load an ELF image, relocate it, provide some additional memory for the new CPU driver to work with, provide a \texttt{core\_data} structure, and then make a system call to boot the new core with a given starting address.

Once you have a system call that starts a second core, you might want to pause the kernel on the BSP-core until the kernel on second core is running and is in a stable state. You can use the communication register mentioned in Step 2 for this purpose. Again, we advise to delay resuming the BSP core as long as possible to avoid concurrency related issues.

You should be able to see prints from the kernel on the second core after completing this step. Your second (or both) kernels may panic initially! But don’t worry, we will work on fixing those issues next :-) You may refer to Samuel Hitz’s thesis (section 4.2 in particular) for details on what needs to be done when you boot-up another core. This work was done on the \texttt{GEM5 Simulator}, and can help you in understanding issues involved in booting Barrelfish on a multicore system.

You can test the stability of your system by booting the application core and let it go as far as possible without crashing while BSP core resumes and spawns new applications in parallel. Ideally, you should be able to reach and partially execute \texttt{arm\_kernel\_startup} on application core while BSP core still starting other applications.

### 4.3 Memory management in presence of two cores

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 different ways to tell the second core which part of memory it is responsible for. You can pass this information as part of the arm_core_data struct which is passed to the cpu driver, or you can statically partition the available physical memory between the two cores. For further discussion we assume a statical partitioning of the memory.

You can find the size of available memory by using the size_ram function in kernel/arch/omap44xx/init.c. You need to understand how exactly the kernel is using memory, and then modify the code so that each kernel will use approximately half of the memory (or some other ratio). Note that you may still want to map all the memory and address-space, but consider only half of it when using it for internal allocations, to load the init ELF image, and when passing it to init for further memory management.

As a starting point, here are few interesting functions that deal with memory management within the kernel:

- bsp.alloc.phys
- create.phys_caps
- init.page.tables
- spawn_init.common
- spawn_module

There are a lot of places where the BSP core gets special treatment because of the way we handle resources, particularly because the BSP is assumed to take care of all the available memory. But with your new design, both cores need to take care of their part of memory. This also means that you need to re-think all the code which differs in execution based on whether it is running on the BSP core or not. The code that we’ve handed out already contains some of the bits required for this change (e.g. the parameter alloc_phys_fn to spawn_bsp_init and the functions called from spawn_bsp_init).

As we are treating both cores as BSP-like cores, both of them should have access to the information which is passed to the BSP core by the boot-loader in the arm_core_data structure. The second core gets its arm_core_data structure from init on the BSP core. The information exchange between boot-loader, kernel and init is done using the arm_core_data and bootinfo structures, and you may need to understand them and how/where they are converted from one form to another. You can modify these structures to pass additional information that you may need in the cpu-drivers and init processes on both cores.

Now, you are all set to start applications on the second core. You should be able to use the modified version of spawn_bsp_init to start init on the second core, and init on the second core should be able to start other applications on that core.

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

5 Communication between two cores

However, just 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 (from Milestone 6) 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 (the shell process in our case).

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 and use it to implement a shell which can start applications on both cores.

5.1 Sharing a frame between two cores

As Barreﬁsh is related to the micro-kernel design philosophy, we prefer pushing most of the functionality out of kernel. This also means that we should try and provide a form of intra-core communication which can be done directly from user-space without involving 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.

So, you should implement a simple shared memory based, user-mode communication channel. There are multiple ways of implementing such a channel. Following is one way to implement it, but feel free to experiment with other designs.

a) Create a sufﬁciently large frame in init.0 (i.e.: init on the BSP core) which can be used for communication. You may want to map this frame as uncacheable to avoid running into caching related issues at cost of performance.

b) Pass information (i.e.: physical address, size) about this frame (lets call it communication frame) to the kernel which will run on application core. This kernel can then pass this information to the init process starting on the application core during the startup time. Now init.1 on the application core can use this information to map and access the same area of physical memory. This way, both instances of init will have access to a region of memory which can be used to communicate requests from init.0 to init.1 about running applications on core 1.

There are multiple ways to pass the frame from kernel.1 (i.e.: the kernel on the application core) to init.1. We can either directly map it into address-space of init.1 at some virtual address and pass the address to init.1 as a command line argument, or create a special capability for this frame in some ﬁxed slot within init’s cspace and let init.1 map this frame within its address-space itself from this well-known slot.

5.2 Establishing a communication protocol

As we now have shared memory between two applications, we should establish a protocol which enables meaningful communication between these applications. 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 then should 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, response 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:

```c
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.

**EXTRA CHALLENGE**

If you’re feeling ambitious you can experiment with forwarding more calls from your existing RPC interface (that is only working for applications on the same core currently) over the x-core channel.

Note: If you decided to split the physical memory between cores you’ll need to make sure that you don’t accidentally forward messages that should be handled by an application on the same core.

Additionally you should implement a oncore shell command that has an arbitrary amount of arguments where the first argument is a core id (0 or 1), the second argument is a program name, and the rest of the argument are arguments to the program specified in the second argument. Find a few examples that illustrate oncore below.

```bash
$ oncore 1 memeater
# starts memeater on core 1
$ oncore 0 hello
# starts a hello application on core 0
$ hello
# starts a hello application on the core the shell is running on
```

Please note that application execution on remote-core should be – by default – asynchronous. Once the application starts running on a remote core, remote spawn should return to the calling process.

**EXTRA CHALLENGE**

If you want to go the extra mile, you can make your system support running the shell on the core where the serial driver isn’t.

**EXTRA CHALLENGE**

Implement more generic communication between processes across cores so that any two processes can setup a direct channel between each other.

**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.

### 6 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 page faults.
• You can run applications on the second core.
• You can start applications that run on the second core from the BSP 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.