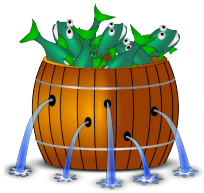

ADVANCED OPERATING SYSTEMS
Milestone 2: Process Creation
Fall Term 2016

Assigned on: **07.10.2016** Due by: **21.10.2016**

In the previous milestone, you built a simple allocator for physical memory running the first process to start, `init`. The `init` process is a special case in that it is the only process started by the OS kernel (in Barrelfish the kernel is known as the *CPU driver*, for reasons that will become clear later).

Operating systems create processes in different ways, though the basic set of steps is common to most. In almost all cases, the first process is special, and is created in a fairly ad-hoc manner by the kernel.

In a Unix-like OS like Linux, all processes after `init` are created in the kernel by forking an existing process. Unix then allows an existing process to be replaced by a new program (generally loaded from the file system) using the `exec` call. This combination of `fork()` and `exec()` is arguable the defining feature of Unix.

In contrast, Windows generally starts a process from scratch, given a binary file to run. Like Unix, however, this is done in the kernel.

In Barrelfish, a new process is created by an existing process which allocates space for all the process data structures (including the process control block, the text segment holding the program code, the stack, data, and bss segments) using the capability facilities we saw last week.

In this milestone, the goal is to start another process in this way. At bootstrap time, the initial process `init` is created and started by the CPU driver. Your task now is to create a subsequent process from `init`.

To create a process, the following high level steps are needed:

- Finding the appropriate ELF image for the program.
- Creating an initial CSpace for the new process, so that it has all the capabilities in needs when it starts to run.
- Creating an initial VSpace layout, so that when the process starts running its virtual address space is sufficiently complete to execute the code and access its data.
- Loading the ELF image containing the program for the new process into both parent and child address spaces.
- Adding a *dispatcher* for the new process, and telling the CPU driver about it so that the new process can be scheduled and dispatched.

Much of the work of creating a new process is memory mapping operations and keeping track of your own process (in the first case, `init` and your child process' virtual address spaces). To do so, you will extend the memory management functions that you have started to use in the previous milestone.

This milestone is due in two weeks. As you can see, creating a process can be a complex business. We stongly suggest starting early!

1 Get Prepared

As a first step, you should obtain any new updates from the git repo, and merge the `week3` branch into your repository.

You might want to have a look at the functions `spawn_load_with_bootinfo` from upstream Barrelfish in the file lib/spawndomain/spawn.c.

2 Implementation

We'll go through the process, as it were, in stages.

2.1 Extend paging.c

In the previous milestone you mapped a frame (i.e. a physical page) into your own virtual address space at a virtual address provided by the caller.

In this milestone, you'll need to extend this code to be able to map frames at a free virtual address (to be allocated) in the virtual address space of a possibly different "domain" (which, for the moment, is the Barrelfish equivalent of a process).

When doing this, you must not change the function signatures as provided in `paging.h`. There is no need to, and in any case they are used in various other places in the code.

If you have not done so, you should complete your implementation of `paging_map_fixed_attr` such that it can map a frame of arbitrary size to a specified location. Keep in mind the structure of an ARMv7-A page table: in particular, note that a large frame region can span more than one L2 page table.

Once you completed this functionality, you will need to implement code to keep track of your virtual address space. In particular, you need to keep track of which virtual addresses are already allocated and which ones are free. For this purpose, you need to implement the function `paging_alloc` in `paging.c` which returns a free virtual address region of the requested size. You can add additional needed state to `struct paging_state`. Your virtual address allocation should start at the address `start_vaddr` provided to the `paging_init_state` function.

Once you have written both functions, you can then combine them to implement `paging_map_frame_attr`, that finds a free virtual address and maps a frame at that address.

2.2 Find and map the ELF binary

We can now move on to loading a program image in an address space.

The code you write for creating a process or domain should be put in the library `lib/spawn/`. Specifically, you should provide an implementation for the function stub `spawn_load_by_name` in `spawn.c`.

The init process will use functionality from `lib/spawn` so you should add the library `spawn` to the Hakefile for the `init` program.

Make sure to include a binary for the process you want to create in your boot image. A good first candidate is the `hello` program that should be already in your source tree. To include it in the PandaBoard boot image, modify the platform Hakefile `platforms/Hakefile` to add it to the list `modules_common`.

In Barrelfish, we adhere to the "multiboot" specification for loading the modules of our operating system. Multiboot is essentially a way to combine a collection of "modules" (think of them as files) into a single boot image, so that code (in this case, `init`) can later extract them individually. The program to run in

your newly-created process will, at this stage, be stored in a multiboot module. Later in the course, you might be able to load programs from the file system instead. All the information about a multiboot image (and, in particular, where all the modules are) is represented in memory using a `struct bootinfo` datastructre.

You can use the multiboot functions defined in `lib/spawn/multiboot.c`. To locate a module by name use the function `multiboot_find_module`. This library maps the multiboot modules in frames, so it will not work correctly unless you have implemented the previous step.

When it creates the `init` process for you, the CPU driver maps the `struct bootinfo` record into your `init's` process virtual address space. The (virtual) address of this record is then passed to `init` as the first command line argument, so you can get a pointer to the bootinfo structure as follows:

```
bi = (struct bootinfo*)strtol(argv[1], NULL, 10);
```

Once you have obtained the `struct mem_region` for your module from the multiboot image you can use the `mrmod_slot` field to create a `struct capref` to the binary's frame.

```
struct capref child_frame = {
    .cnode = cnode_module,
    .slot  = module->mrmod_slot,
};
```

At the end of this step you should be able to verify that the first four bytes of your mapped binary contain the bytes `0x7f 'E''L''F'`. This is the ELF "Magic Number" - all ELF files start with this sequence.

2.3 Setup initial C- and V-Space

We now need to create a CSpace (i.e. the list of capabilities) and a VSpace (a virtual address space, in other words a page table) for the new domain.

The CPU driver expects the CSpace to be represented as a two level tree, so you will have to create both a Level1 CNode and a couple of Level2 CNodes that are linked in the L1 CNode for your child process - this should give you enough capability slots for the time being.

2.3.1 Creating CNodes

To create the child's CNodes you can use the following functions:

```
errval_t cnode_create_l1(struct capref *ret_dest, struct cnoderef *cnoderef);
```

This function creates a new Level1 CNode and fills in a capability that references this CNode in `ret_dest`.

```
errval_t cnode_create_foreign_l2(struct capref dest_l1, cslot_t dest_slot,
                                 struct cnoderef *cnoderef);
```

This function creates a Level2 CNode. In comparison to the `cnode_create_l2` function that you've encountered in the previous milestone, this version allows you to specify the L1 CNode you want to use (for example the one you've already created for your new process using `cnode_create_l1`) - that's why it's a "foreign" CNode

You can create a `struct capref cap` that references a slot in the Level2 CNode by setting `cap.cnode = cnoderef`, where cnoderef is returned from the L1/L2 create function.

2.3.2 Create the CSpace

Next, we need to populate the new process' CSpace with useful capabilities.

The newly created process must be able to find and use some of the capabilities that are created during process creation. For example, the child process must be able to find the L1 pagetable capability so that it can map in memory into its own address space.

To do so, we create the CSpace with *conventions* about how it is laid out, and which capabilities are where. You can use the constants in `include/barrelfish_kpi/init.h` to determine the location of the fixed capabilities.

The format looks as follows; note that the Barrelfish CPU driver refers to a schedulable unit as *dispatcher* - this will represent your process to the scheduler. Your process will expect the following Level2 CNodes:

a) TASKCN: Contains information about the process itself.

- SLOT_SELFEP: Endpoint to itself. You can get this capability by retyping the dispatcher capability to ObjType_EndPoint. Endpoints are used for inter-dispatcher communication, and you'll use them in the next milestone. Don't worry about that for now, though, other that creating SELFEP as described here.
- SLOT_DISPATCHER: Contains the dispatcher capability. The dispatcher is the equivalent of a "Process Control Block" in other operating systems. You can create one using `dispatcher_create(...);`
- SLOT_ROOTCN: Contains a capability for the root (L1) CNode.
- SLOT_DISPFRAME: See the next section for an explanation of this.
- SLOT_ARGSPG: A page containing a list of command line arguments.

b) SLOT_ALLOC_0: Empty L2 Node that contain space for the child's initial slot allocator. When the new process starts creating and retyping capabilities, they will be stored in here.

c) SLOT_ALLOC_1: A second L2 Nodes for the slot allocator to use.

d) SLOT_ALLOC_2: A third L2 Nodes for the slot allocator to use.

e) SLOT_BASE_PAGE_CN: Each slot holds a BASE_PAGE_SIZEd RAM capability.

f) SLOT_PAGECN:

- Slot 0: Contains a capability for the process' ARMv7-A L1 pagetable.
- Other slots: Used to store ARMv7-A L2 pagetables and mapping capabilities.

2.3.3 The initial VSpace

Having created the initial capability set for the new domain, you now need to construct its virtual address space. This means creating hardware page tables, and populating them with mappings to physical memory.

To create a new Level1 Pagetable, you can pass `ObjType_VNode_ARM_l1` to the following function.

```
errval_t vnode_create(struct capref dest, enum objtype type)
```

Now you can use the same function to create mappings as you have used in Milestone 1 (`vnode_map` and friends).

Keep in mind that you can only invoke a capability that is in your own CSpace. To invoke a capability that resides in the child's CSpace, you must first copy it in the parents CSpace using
`errval_t cap_copy(struct capref dest, struct capref src)`.

2.4 Parsing the ELF

You need to look into the binary file extracted from the multiboot image to figure out information like where it needs to be located, which memory segments need to be created, etc.

You can use the `elf` library in the source tree to parse the binary. The function `elf_load` takes a pointer to the (mapped) binary and calls a callback function for each section that should be created. For `em_machine` you should use the constant `EM_ARM`. `base` and `size` are the address and size of the mapped binary. `retentry` will be filled with the entry point of your child process.

Your callback function must return a pointer in `ret`, so that the elf library can copy in the data.

```
errval_t elf_load(uint16_t em_machine, elf_allocator_fn allocate_func,
                  void *state, lvaddr_t base,
                  size_t size, genvaddr_t *retentry);

typedef errval_t (*elf_allocator_fn)(void *state, genvaddr_t base,
                                     size_t size, uint32_t flags, void **ret);
```

You will also need to find the `.got` (global offset table) section and use the address to initialize the offset registers. The GOT stores the absolute addresses of all global variables and is located at a fixed offset from the code. With this table another layer of indirection is added to enable the use of shared libraries (which reside at different addresses for different processes) . To locate the section use the function:

```
struct Elf32_Shdr* elf32_find_section_header_name(genvaddr_t elf_base,
        size_t elf_bytes, const char * section_name);
```

2.5 Set up the dispatcher

In the Barrelfish CPU driver, all information about a process (on a core, effectively a dispatcher) is represented by a capability type called a dispatcher frame.

It is created by the spawning process and then given to the CPU driver. The CPU driver will use this memory area to store information about the process. For example, it will save information like register state on a context switch, or the error handlers to be called on page faults.

The size of the dispatcher frame is given by `1<<DISPATCHER_FRAME_BITS`. You should allocate a frame of this size and map it into both `init`'s and the child's address spaces. The capability for this frame should also be stored in the child's CSpace in the appropriate slot.

The dispatcher is divided into two structs. One contains architecture-independent members `struct dispatcher_shared_generic` and one contains ARM-specific fields `struct dispatcher_generic`. Then there are two register states `enabled` and `disabled`, which are used to implement the "scheduler activations" upcall we talked about in the lecture, and help with user level thread-scheduling. The process should start in `disabled` mode. To get access to these structs you can use the following code, assuming you have a pointer to the mapped dispatcher frame stored in `handle`:

```
struct dispatcher_shared_generic *disp =
        get_dispatcher_shared_generic(handle);

struct dispatcher_generic *disp_gen = get_dispatcher_generic(handle);

struct dispatcher_shared_arm *disp_arm =
        get_dispatcher_shared_arm(handle);

arch_registers_state_t *enabled_area =
    dispatcher_get_enabled_save_area(handle);
```

```
arch_registers_state_t *disabled_area =
    dispatcher_get_disabled_save_area(handle);
```

We need to fill in some initial information in the frame so when the CPU driver will try to switch to the program for the first time, it will not immediately crash. The information you must fill in is as follows:

```
disp_gen->core_id = ... ; // core id of the process
disp->udisp = ... ; // Virtual address of the dispatcher frame in childs VSpace
disp->disabled = 1; // Start in disabled mode
disp->fpu_trap = 1; // Trap on fpu instructions
strncpy(disp->name, ..., DISP_NAME_LEN); // A name (for debugging)

disabled_area->named.pc = ...; // Set program counter (where it should start to execute)

// Initialize offset registers
disp_arm->got_base = ...; // Address of .got in childs VSpace.
enabled_area->regs[REG_OFFSET(PIC_REGISTER)] = .. ; // same as above
disabled_area->regs[REG_OFFSET(PIC_REGISTER)] = .. ; // same as above

enabled_area->named.cpsr = CPSR_F_MASK | ARM_MODE_USR;
disabled_area->named.cpsr = CPSR_F_MASK | ARM_MODE_USR;
disp_gen->eh_frame = 0;
disp_gen->eh_frame_size = 0;
disp_gen->eh_frame_hdr = 0;
disp_gen->eh_frame_hdr_size = 0;
```

2.6 Set up arguments

Next, we deal with the command-line arguments to the new process. If the process is started by an actual shell (and may be later in the project), the shell needs a way to pass such arguments to the new process.

At the beginning of the arguments page (i.e., the frame in `SLOT\_ARGSPAGE`), the initialization code assumes that there is a `struct spawn_domain_params` lying there. The child's startup code expects everything that does not explicitly have to be filled in by `init` to be zeroed. The remainder of the page (after `struct spawn_domain_params`) is used to store your command line arguments. Make sure that no pointer stored inside your `spawn_domain_params` points into `init`'s address space, but instead all should refer to the childs address space. Also, make sure that after the valid `argv` and `envp` pointers, there is a NULL pointer to signify the end of the list.

A nice thing to do is pass on the arguments from the multiboot specification so that the arguments for your new process can be specified in the `menu.lst` file. You can access these arguments using the `multiboot_module_opts` function.

2.7 Start the process!

Finally, you can now make your dispatcher runnable by invoking the dispatcher capability using the function `invoke_dispatcher`. As `domdispatcher` you can pass `cap_dispatcher` that is statically defined.

If you've done everything right, the child process should be able to print a message. If so, congraulations, you've created a process and successfully started it!

Hints

The following hints might be useful:

- Your init process has only a 64kb stack. Make sure that you do not put any big data structures there - use the heap.
- It's okay to use `malloc` and `free` for your implementation. But, be aware that the libaos provides init only with a static heap size of 16 MB (see `lib/aos/morecore.c`). If this isn't enough, you can use the slab allocator you already encountered in the previous milestone.
- ARMv7-A Level1 pagetables must be aligned on a 16kb boundary. Make sure your `ram_alloc_aligned` ensures 16k alignment.
- After your child process terminates, the kernel will print "revoking dispatcher failed", don't worry about that (yet).

THE MILESTONE

- Fully implement `paging_alloc` and `paging_map_frame_attr`.
- Show that you can start multiple child processes by printing a message from the new processes.

CHALLENGES

- Your child process is unaware of its `struct paging_state`. A simple way to avoid running into trouble with such a limitation is to prevent mappings within certain regions of the virtual address space. Alternatively, to allow arbitrary mappings in the child, the complete paging state must be passed to the child. Implement a method for passing the `struct paging_state` to the child. (1 bonus point)
- Implement the unmapping functions such as `paging_region_unmap`. (2 bonus points)

ASSESSMENT

- Show your implementation of `paging_map_frame_attr` is correct by mapping a large frame.
- Show that you mapped in the ELF binary correctly by printing its magic number.
- Be able to explain how you set up the CSpace and VSpace and where the childs capabilites are stored.
- Show that you can start a process by letting the new process print a message.
- Start the process multiple times.