In this milestone, you will implement a page fault handler as the core component for self-paging user processes. You will also see and use for the first time capabilities as a mechanism for identifying regions of physical memory – these will be covered in more detail in Thursday’s lecture. The reading material for self-paging is already up on the course webpage.

You won’t be building complete demand paging to disk (or some other storage device at this stage), but you will have to handle page faults. The goal for this week is to perform lazy allocation of the program heap (the memory allocated by malloc): instead of making sure that all the physical memory for the heap is there at the start of the program, you will simply reserve the virtual memory, and then fill it on demand with physical memory in response to page faults in that area of the address space.

1 Step 1: Getting prepared

For this milestone you will get a full Barrelfish cpu driver and a somewhat more complete init process, as well as some support libraries (libc, a cut down version of libbarrelfish – our OS library, libmm, libmdb).

You can get the new source tree as follows

```bash
mkdir -p ~/aos-part2
cd ~/aos-part2
tar xf /pub/aos/handout/m2.tar.gz
```

Also, the compilation process has changed slightly because we have renamed the target you need from aos_image to pandaboard_image. Thus you will need to build the image the following way in a empty directory.

```bash
mkdir -p /local/<nethz-id>/build-m2
cd /local/<nethz-id>/build-m2
~/aos-part2/hake/hake.sh -s ~/aos-part2 -a armv7
make pandaboard_image
```

Note that there is a configuration variable called milestone in hake/Config.hs in the build directory. You should always update that variable to the number of the milestone you’re currently working on as this will change the behaviour of some of the code we provided to you in this milestone.
Running the image resulting from an unmodified source tree should give the following output

```
<lots of kernel output>
kernel[0] Calling dispatch from arm_kernel_startup, start address is=204000
init: invoked as: init_memtest 2097152
static malloc buf filled
static malloc buf checked
did not get a buffer
Aborted
```

1.1 EXTRA CHALLENGE

From this point on in the course, you’ll be using the regular Barreelfish kernel (or CPU driver, as we call it). We’ll offer extra points for every bug that you can find and fix in our code!

2 About Capabilities

In this assignment you will use Barreelfish’s capability system to get physical memory for handling page faults. The capability system is Barreelfish’s way of tracking and authorizing access to memory (and, as we will see, many other physical resources in the system) to a user program.

A capability can be thought of as a special kind of handle or pointer. Capabilities cannot be forged, but can be passed around between applications.

For now, it suffices to know that there are different types of capabilities for different types of memory, such as page tables or user-accessible (mapped) memory.

The types you will need in this assignment are the following (see capabilities/caps.hl for a full list of Barreelfish’s capability types):

- ObjType_Frame (for pages)
- ObjType_VNode_ARM_l2 (for ARM level 2 page tables)

Barreelfish stores each process’ capabilities in slots in that process’ capability space (cspace). Each cspace has a hierarchical structure (a bit like a page table itself) that is constructed on demand.

There are also a number of functions that allow you to request new capabilities of these two types. Use

```
frame_alloc(struct capref *ret, size_t bytes, size_t *retsize)
```

to allocate Frame capabilities that can be mapped into an address space and

```
arml2_alloc(struct capref *ret)
```

to allocate capabilities for ARM level 2 page tables.

These functions use the capability slot allocator and RAM allocator to create the requested capabilities. These allocators have a few tricky interdependencies and therefore need special handling while booting up the system.

The `struct capref` type is what we use to reference a capability slot (and hence the capability in it) in user space. It is a unique identifier of one address (i.e. one slot) in the process’ capability space.
We need to make sure that applications cannot construct arbitrary page tables – instead, we wish to ensure that an application can only put an entry in a page table if it refers to a physical frame that it has already been granted use of.

To enforce this, we handle the updating of page table entries using the capability system. We can invoke capabilities as if they were objects – essentially calling a function associated with the capability’s type – and we use these invocations (among other things) to modify page table entries. The high level interface for writing a page table entry looks as follows:

```c
vnode_map(struct capref dest, struct capref src, capaddr_t slot, uint64_t attr, uint64_t off, uint64_t pte_count)
```

The `slot` argument indicates the page table entry we want to modify, `attr` are the mapping attributes (you probably want `VREGION_FLAGS_READ_WRITE`), and `off` is an offset into the `src` memory region. The last parameter is there to indicate the number of pages you’d like to map if you’re mapping a Frame capability of more than 4k in size.

Also, the Barrellfish CPU driver enforces that each copy of a capability is mapped at most once, so you will have to use

```c
cap_copy(struct capref dest, struct capref src)
```

to create copies if you’re working with large Frame capabilities.

You’ll notice that this function looks a bit generic, even though you’ll only be using it for a couple of use-cases. One reason the arguments are generic is that we can use the same function to map L2 page tables in the L1 page table and pages in the L2 pagetables. The other reason for this will be seen later on in the course - Barrellfish can actually construct any page table for any processor architecture this way, even on a different architecture to the page table.

Barrellfish stores the capability for the L1 pagetable at a well-defined location in the capability space, namely slot zero of the `page cnode`. You can create a `struct capref` to that location by hand as follows:

```c
struct capref l1_pagetable = {
  .cnode = cnode_page,
  .slot = 0,
};
```

and use that `struct capref` as destination parameter for `vnode_map()`. `cnode_page` is defined in `lib/barrelfish/capabilities.c`.

### 3 Step 2: Implement an exception handler

In order to handle page faults you need to have a way to run custom code when an exception occurs. This can be accomplished by setting up an exception handler. You can set an exception handler for a thread using the function `thread_set_exception_handler` in the threads package in `libbarrelfish`.

As a first step it might make sense to just print something when you get an exception and stop.

### 4 Step 3: Designing the address space layout

This step should sound familiar, because you have already done something similar for kernel page tables. The new CPU driver uses the same half-and-half split of the address space where the lower 2GB of the address space are for the user process and the upper 2GB are for the kernel.
However, what you need to think about this time is how to represent the address space at user level, so that your paging code has knowledge about occupied and free virtual address regions. This will require quite a bit of C programming for which we won’t give you much of a template, but you will need to create data structures that maintain a 2-way mapping between physical frames and virtual addresses for the application. You can do this any way you like (a list, a tree, a hash table, etc.) but keep in mind which operations need to be efficient at runtime, and be prepared to explain your choice in the marking session.

Note: you should have all the state of your implementation in the `struct paging_state` of which you’ll find an instance called `current setup` as a global variable in `lib/barrelfish/paging.c`. Also, have a look at the different comments that are marked TODO in `lib/barrelfish/paging.c` to see what types of functionality you will need in later exercises.

4.1 Step 3.1: Implement page fault handling

You should now implement handling of page faults in your exception handler. You should also think about detecting possible NULL pointer dereferences and disallowing any mapping outside the range(s) that you defined as valid for heap, stack, etc.

4.2 Step 3.2: Implement virtual address allocation

In order for `libbarrelfish` to work properly, you’ll need to implement the function `paging_alloc` in `lib/barrelfish/paging.c` to return suitable addresses according to your current address space occupation.

4.3 Step 3.3: Dynamic memory allocation

For this step you should implement `morecore`, which is the backend for `malloc` and friends. You should do this without resorting to having a large static array (as the current code you have does).

Instead, you should now be able to just reserve a region of virtual addresses as the heap (using `paging_alloc`) and return one of these addresses as new heap space for `morecore`. Following this, the application can handle page faults, and have the page fault handler allocate physical memory on demand to back the reserved virtual address space.

4.4 Notes

You may get the following assertion failure:

```
assertion "state->base_capnum < OBJSPERPAGE_CTE" failed:
file "../solution/lib/barrelfish/ram_alloc.c", line 100,
function: ram_alloc_fixed
```

If you do, then you have run out of space to store capabilities. This is most likely the case if you allocate a 4 kilobyte Frame capability each time you have to handle a page fault.

An easy way to fix this is to allocate larger Frame capabilities and then map individual 4 kilobyte chunks of these capabilities when a page fault occurs.
THE MILESTONE

- Implement self-paging on top of a full Barrelfish cpu driver
- Show your implementation handling page faults

CHALLENGES

- Handle the case of running out of free space in the cspace

ASSESSMENT

- Init taking and handling page faults
- Explain user-level address space storage format
- Malloc without static memory allocation
- Your implementation should be able to handle dynamic allocations of total size of at least 32MB.