Exercise #1: Boot

The purpose of this exercise is to learn how to setup and boot a JX system.

a) Checkout the sources from the repository.

b) Write a boot.rc file that boots a configuration with the name [AKBP2]. This configuration should start the following domains:

- "BlockIORAM"
    - Heap size: 2000000
    - Component: "bio_ram.jll"
    - Start class: "bioram/BlockIORAM"
    - Parameters: "BIOFS"
- "FS"
    - Heap size: 40000000
    - Library: "test_fs.jll"
    - Start class: "test/fs/FSDomain"
    - Parameters: "BIOFS", "FS"
- "IOZONE"
    - Heap size: 8000000
    - Library: "test_fs.jll"
    - Start class: "test/fs/IOZoneBench"
    - Parameters: "FS", "4", "8192"

The `BlockIORAM` class implements a RAM disk device. It registers the device under the name that is passed as a parameter ("BIOFS"). The `FSDomain` class uses the block I/O "BIOFS" for a file system. It registers the file system under the name "FS". The `IOZoneTest` class uses this portal to run the iozone benchmark for file sizes 4 KB to 8192 KB.

c) Activate the monitor and explore the system state. What domains and threads are running, what objects are on the heap of each domain, what threads are in the run queue? Which portals are registered at the kernel name service?

Exercise #2: Hello World!

During this exercise you will create your own library and start a domain with this library.

a) Add a new library with the name `hello_akbp`.

b) Write a start class with the name `akbp2.HelloWorld` that prints "Hello World!" to the debug output.

c) Create a new configuration in `boot.rc` to boot your `HelloWorld` class. Boot the system and wait for "Hello World" to appear on the debug output. Activate the monitor and explore the system state.

Exercise #3: Portals

This exercise introduces you to portals.

a) Write a portal interface with the name `akbp2.FirstService` and a method `void hello()`.

b) Add a new class `akbp2.FirstServiceImpl` which should implement the `akbp2.FirstService` portal interface. The `hello()` method should simply print "Hello World!" to the debug output.

c) Add a new class `akbp2.StartFirstService` to your `hello_akbp` library. This class should be as an entry point for a domain. The domain init method should create an instance of a `akbp2.FirstServiceImpl` and register it at the name server under the name "AkbpSvc".

d) Add a new class `akbp2.StartFirstClient` that looks up the "AkbpSvc" portal and calls the `hello` method.

e) Create a new configuration in `boot.rc` which starts a domain with the `akbp2.StartFirstService` class and a domain with the `akbp2.StartFirstClient` class. Boot the system and wait for "Hello World" to appear on the debug output. Activate the monitor and explore the system state.

f) Create a class `akbp2.ImportantData` which contains no methods and two instance variables `int age` and `String name`. The `ImportantData` class should contain four methods: `void setAge(int age)`, `int getAge()`, `void setName(String name)`, `String getName()`. Add a method `hello2(ImportantData data)` to the `akbp2.FirstService` interface and the `akbp2.FirstServiceImpl` class. The `hello2(ImportantData data)` method should print the contents of the `data` object to the debug output in an endless loop. Change `akbp2.StartFirstClient` to invoke `hello2` . In another thread, the client should change the `data` object that is passed to `hello2`. Boot the system and find out whether the server can observe the change of the object state. (Hint: Use `Thread.yield()` in the endless loops to allow other threads to run in a non preemptively scheduled system.)

g) Create a new portal interface `akbp2.SecondService` and an implementation class `akbp2.SecondServiceImpl`. The portal interface and its implementation should contain four methods: `void setAge(int age)`, `int getAge()`, `void setName(String name)`, `String getName()`. Add a method `hello3(SecondService svc)` to the `akbp2.FirstService` interface and the `akbp2.FirstServiceImpl` class. The `hello3(SecondService svc)` method should print the contents of the `svc` object to the debug output (using the setter/getter methods) in an endless loop. Change `akbp2.StartFirstClient` to invoke `hello3` and change the `svc` object instead of the `data` object. Boot the system and find out whether the server can observe the change of the object state.

Exercise #4: Memory objects

During this exercise you will learn to cope with memory objects.

a) Add a method `hello4(Memory buf)` to the `akbp2.FirstService` interface and the `akbp2.FirstServiceImpl` class. The `hello4(Memory buf)` method should print the contents of the memory object to the debug output (use the `jx.zero.debug.Dump` class from the lib `zero_misc`) in an endless loop. Change `akbp2.StartFirstClient` to invoke `hello4` and change the contents of the memory object in an endless loop. Boot the system and find out whether the server can observe the change of the memory object's contents.

b) The `hello4` method of should revoke access to the received memory object before printing its contents. What happens at the client side?

c) The server should now provide a method `Memory getBuffer(int size)` to obtain a buffer and a method `void processBuffer(Memory buf)` to process a buffer. The client calls getBuffer to obtain a buffer, fills it with data, and calls processBuffer to let the server process the buffer. To process a buffer the server must prepend header data to the buffer it received in the processBuffer call (for example a protocol header). To avoid copying the data, the getBuffer method must create a memory object that is large enough to hold the data of the client and the additional header data. To the client it must return only the memory subrange that excludes the header. In the processBuffer method the server must join the passed buf memory with its predecessor.

Exercise #5: Components

During this exercise you learn how to modularize your code into components.

a) Separate your server and your client into three components:
   - A component that contains the server implementation: `akbp_server_impl`
     The name under which the server registers its portal should be passed as a parameter to the server start class.
   - A component that contains the interfaces necessary to access the server: `akbp_server`
   - A component that contains the client code: `akbp_client`
     The name under which the client tries to look up the server portal should be passed as a parameter to the client start class.

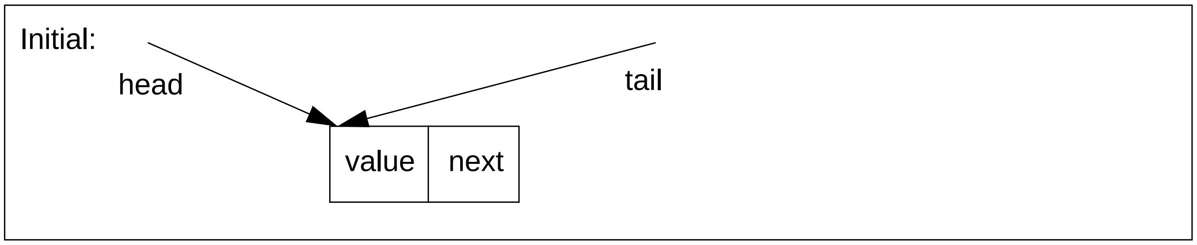
   Change the `boot.rc` file appropriately and boot the system.

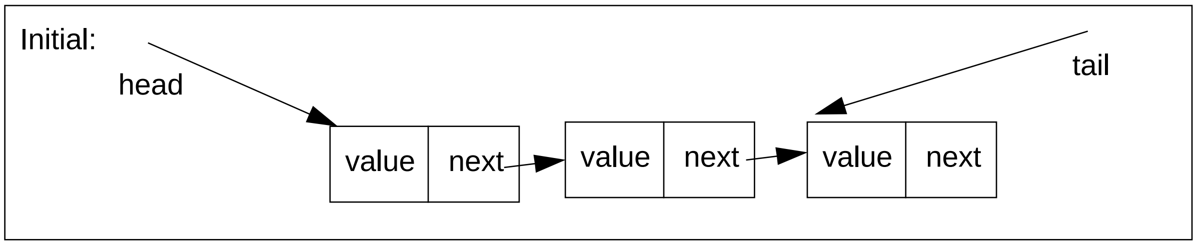
b) Use the `ApplicationStarter` to start your domains.

c) Use the `MonolithicApplicationStarter` to start your system in one domain. Repeat the test of (Exercise #3:.f). Is there a difference in the observed behavior of the system?

Exercise #6: AtomicVariable

a) Implement a single-consumer single-producer queue using a linked list. The producer adds nodes to the tail of the queue and the consumer removes nodes from the head of the queue. There is at least one node in the queue to separate consumer and producer.





The nodes of the linked list should be of type `Node` and contain the fields `Object value` and `AtomicVariable next`. The linked list should be implemented in a class `Queue`, use two fields `Node head` and `Node tail`, and provide two methods: `void enqueue(Object value)` and `Object dequeue()`.
The `enqueue` method should create a new Node and initialize its value. It should then set the next field of the queue's tail to the newly created node and unblock a consumer thread in an atomic operation. It should then set the tail reference to the newly created node.
The dequeue method should block if the next field of the head node is null. Otherwise it should should return the value of the head's next node and set the head reference to this node.
Test your Queue by creating a consumer and a producer thread. The producer thread should add Integer objects to the queue and the consumer thread should check whether they are de-queued in the correct order.

Exercise #7: Timer

During this exercise you will learn to use the timer service.

a) Use the timer to print "Hello World!" each second.

Exercise #8: UDP

During this exercise you will be introduced to the networking subsystem.

a) Send UDP packets.
b) Receive UDP packets.
c) Send a UDP packet each second.
d) Measure UDP round-trip time.

Exercise #9: System inspection

During this exercise you will learn to use the system inspection tools.

a) Create thread activity diagram for the system that sends UDP packets.
b) Compile the system with sampling support. Sample the UDP ping. Where spends the system most of its time?