Exercise 10: QNX Resource Manager

In this exercise we will continue with QNX. This week we will create simple resource managers, and communicate with it both locally and from one computer to another. The IPC messages we used last week used the PID of the server to contact it. Although we solved this with the use of shared memory, there is another solution that is arguably more elegant. This is resource managers, which is similar to device drivers in Linux. Where the device driver is a kernel module, the resource manager is a user-space program (microkernel, remember?).

If you are not familiar with resource managers (or at least have experience with Linux device drivers), please read the information about QNX resource managers, given in the link below.

http://www.qnx.com/developers/docs/6.5.0_sp1/index.jsp?topic=%2Fcom.qnx.doc.neutrino_resmgr%2Fabout.html

1. Resource Managers

A resource manager is represented by one or more files in the filesystem, normally located in the `/dev` folder. The client can communicate with the resource manager by opening, reading and writing this file.

To save you some time, a basic resource manager that creates a device `/dev/myresource` is included in "resmgr.c" on It's Learning. It also includes an `io_read` function which will return "Hello World". Create a new project with this code, compile it and run it on the target. Open the terminal on the target, and verify that there is a `/dev/myresource` file there. Run the command `cat /dev/myresource`, and you can see that it will return "Hello World". You can also read from the resource manager with the "read_file_example.c"-file on It's Learning (create a separate QNX C Project). Confirm that you can both read using cat and the example file.

Assignment A:

This resource manager is obviously not very useful, so you should extend it so you can interact with it with `cat` and `echo` commands properly on the terminal (read/write). The resource manager should have the following specifications:

- The resource manager should contain a char array (can be fixed size and global for simplicity) where you can store a sentence.
- Reading the device (`cat /dev/myresource`) should return the content of the char array.
- Writing a sentence to the device (`echo sentence > /dev/myresource`) should store this sentence in the char array (overwrite the old one).
- The char array in the resource manager should be protected by a mutex (use standard pthread mutexes).

Documentation for how to get your resource manager to react to read and writes can be shown here:

http://www.qnx.com/developers/docs/6.5.0_sp1/topic/com.qnx.doc.neutrino_resmgr/read_write.html

It is not easy to follow these instructions, as many commands are necessary to implement proper read and write functionality for a resource manager. Not all of these are strictly necessary. In 1.1 and 1.2, I

have tried to give some tips for developing the resource manager and how you can simplify the code. It is recommended that you use use "resmgr.c" from It's Learning as a base.

1.1. Overriding `io_read()`

The "resmgr.c" code implements `io_read()`. It shows that you do not need to implement everything from the documentation in order to get your resource manager to work. The following bullet points explains the `io_read()` function implemented in "resmgr.c".

- `iofunc_read_verify()`, the testing of xtype and to change the `ocb→attr→flags` are not needed.
- The example in the documentation is prepared for a partial read, meaning that the client asks to read fewer bytes than what the resource manager intends to send.
    - This is not necessary in our application, because the cat command ask for a large number of bytes at the time, so we can assume that it always will read our entire buffer.
    - We still need to keep track whether the client have already read our buffer, in that case we must return zero bytes.
    - The `ocb->offset` value will start as 0, but if you change it, it will keep the value for the next read. Normally this value is used to keep track of how much of the buffer that has been read, but we can instead set it to 1 to indicate that the device has already been read.
    - If the offset value is 0, the function should return data, if it is 1, the function should return zero bytes.
- You send data with the `SETIOV()` macro, using the buffer as the second parameter and the length of the buffer as the third parameter.
- You set the number of read bytes with the `_IO_SET_READ_NBYTES()` macro. The second parameter should be the length of the data if the function returns data or 0 if it doesn't.
- It returns `_RESMGR_NPARTS(1)` if the function returns data or `_RESMGR_NPARTS(0)` if it doesn't.

You will need to alter the `io_read()` in order for this program to work as intended:

- Create local a buffer (can be fixed size) containing the data you read.
- Protect the resource manager's char array with a mutex.

1.2. Overriding `io_write()`

It is recommended that you implement `io_write()` the following way:

- Start simple, just create the `io_write()` function and override `io_funcs.write`. The function you created should then run each time something writes to the device, and you can test this by adding a `printf()` to the function.
- Do not care about the `iofunc_write_verify()`, the testing of xtype and to change the `ocb->attr->flags`.
- You can define a large char array instead of allocating it with `malloc()`.
- What you should do:
    - Set the number of bytes that the client's write command should return.
    - Use `resmgr_msgread()` to get the written data.
    - Return `_RESMGR_NPARTS(0)`.
- Protect the resource manager's char array with a mutex

2. Counting resource manager

Assignment B:

In this assigment we alter the resource manager to contain a counter variable that can be controlled by writing (`echo`) to the resource. The resource manager should have the following specifications:

- The resource manager should contain a counter variable that starts at zero and can be counted upwards or downwards.
- Reading the device (`cat /dev/myresource`) should return the value of the variable.
- Writing a command to the device (`echo command > /dev/myresource`) should tell the resource manager what to do with the variable.
    - "up" means that the variable value should count upwards.
    - "down" means that the variable should count downwards.
    - "stop" means that the variable should stop counting.
- It should have a thread that can increase or decrease the variable every 100 ms (use `delay(int ms)`).
- Any global variables in the resource manager should be protected by a mutex (use standard pthread mutexes).

3. Qnet

A resource manager can not only be accessed by local processes, it can also be accessed by processes running on other machines over Qnet. To enable Qnet, we must create a file in the `/etc` folder that tells QNX that Qnet should run. You can use the following command:

```
touch /etc/system/config/useqnet
```

We can easily create another QNX target in VirtualBox, by right-clicking our target while it is not running and select "Clone". You will then get an identical computer. After cloning, you must enter the settings of the clone, select "Network" and in the advanced options you should click on the refresh button that creates a new random MAC address. The clone must have a different MAC address than the first target.

You now have two targets, and should start both. When they have booted, you should select "Configure->Network" on both targets. Select two different host names for the targets. Apply (can take a while) and reboot.

When rebooted, confirm that the two machines have different IP addresses. Then you can run `ls /net` on one of the computers, and you should see the two host names. From this folder you can access the file system from one computer to the other.

Assignment C:

Run the counting resource manager on one of the computers, and access it with `cat` and `echo` commands from the other computer using the `/net` folder.

4. Blocking I/O

There are situations where the resource manager doesn't want to reply to the client right away. This is typically because there is currently no available information for the client. In these situations the client can specify whether it wants to wait until information is available or if it doesn't want to be blocked by using the O_NONBLOCK flag when opening the device.

Assignment D:

In this assignment we will create a resource manager that can block clients on reading requests. Create a resource manager that does the following:

1. When the device is written to, the message should be stored in a queue. If the queue is full, the information can be discarded.
2. When reading the device, the oldest message in the queue should be returned and removed from the queue.
    - If the queue is empty and the client uses the O_NONBLOCK flag the read should return nothing.
    - If the queue is empty and the client does not use this flag, it should block the client until a message is received and it is returned.
3. You can use the echo command to write to the device, but should create a simple application that reads from the device so you can easily switch between blocking and non-blocking and see the difference in behavior. It is not recommended to use `cat`.
4. You have to search and figure out for yourself how blocking/non-blocking in QNX works. The tutorial linked in assignment A should provide the necessary information.
5. You need to somehow make sure that it is impossible for two threads to read/write to the resource manager at the same time. How you solve this is up to you.

For the queue you can write your own or use the supplied FIFO on itslearning.

5. Approving the exercise

For this exercise you need to show the following to a student assistant to get the exercise approved.

Show your counting resource manager and how you can interact with it using `cat` and `echo` commands, both locally and on from the other target.

Show how the resource manager in assignment D differs when reading blocking vs non-blocking.