Exercise 8: QNX IPC

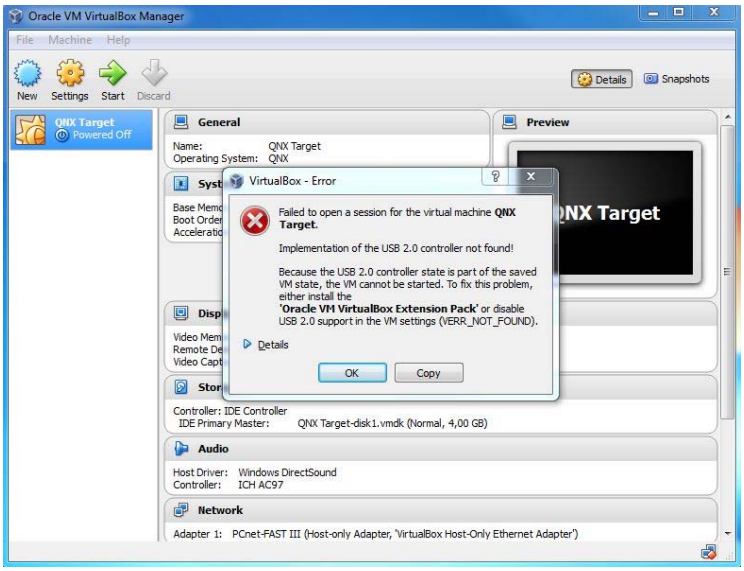
In this exercise we will use the real-time operating system QNX. It is also a UNIX-like system, which means that much of the programming will be similar as for Linux. QNX programs is normally developed using a host-target configuration, where the host computer can have any operating system. We will use Windows on our host system, and will program virtual QNX targets that run on VirtualBox. VirtualBox is a program that lets you run a virtual computer inside your main computer, inside its own window.

QNX is a microkernel operating system, and only some fundamental functionality runs in the kernel. The rest of the functionality runs as user-space processes, and the communication between these becomes very important. In this exercise we will look at two methods processes can communicate, shared memory and messages.

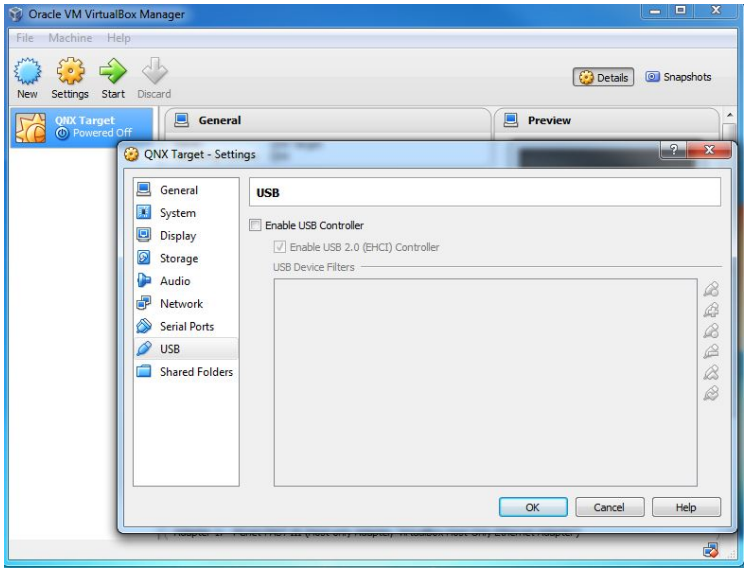
1. Starting the QNX target on VirtualBox

Download the file "QNX Target.ova" from the link on it's learning. Open the file in VirtualBox, and you will get an import wizard. You don't need to do anything here, but you can change the name of the computer if you like. Importing should take a few seconds, and you will then get the VirtualBox main window, with your new QNX computer in the list.

Start your QNX target by double-clicking it. You should see the computer booting in a window. It you receive the following error message



make sure to disable the USB controller in the target settings as displayed in the following picture.



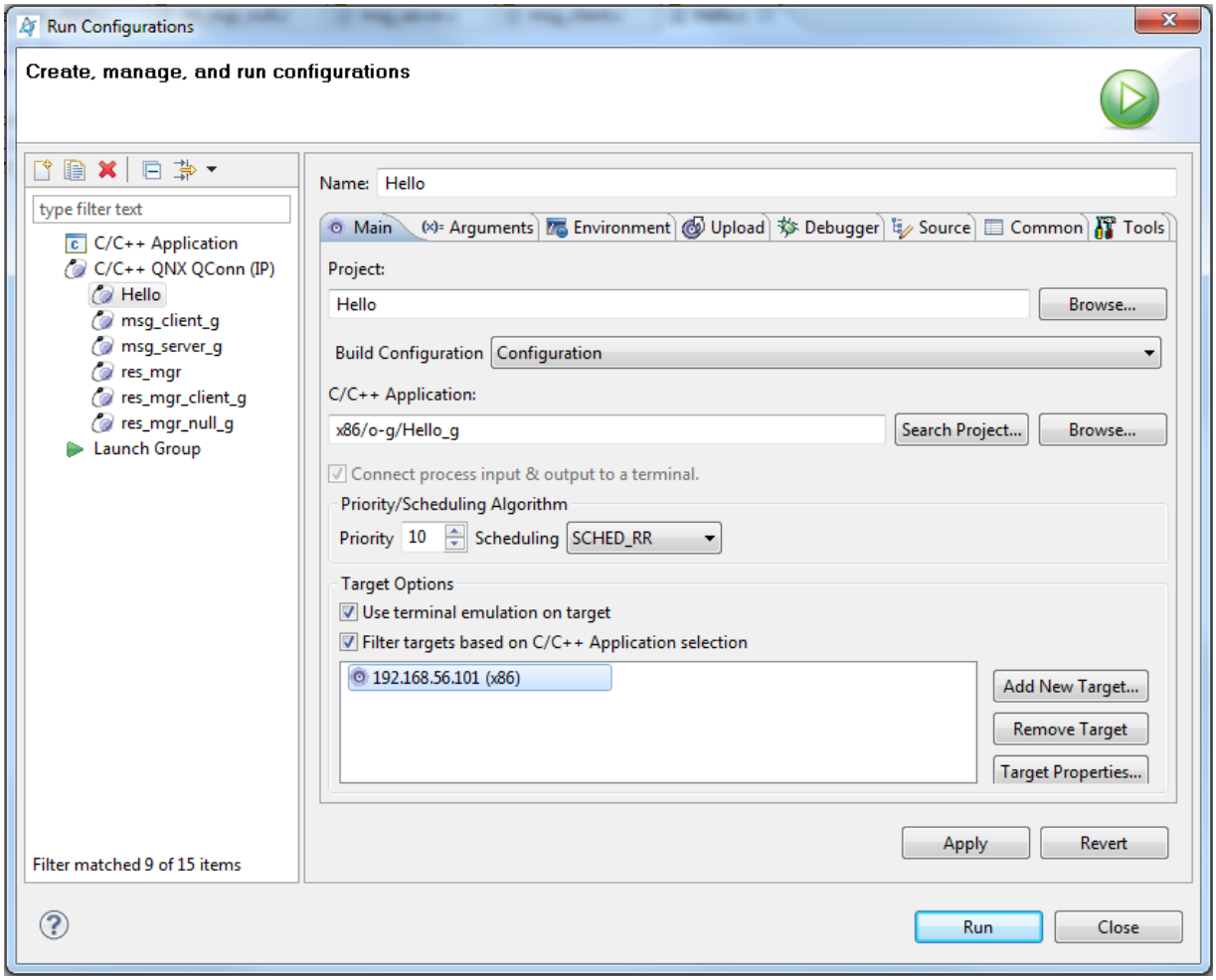
There will be several popup windows during boot, and you can safely say yes to all of these. You should login as root, with a blank password. When you click the QNX window, your mouse will be locked inside the virtual computer; to escape you can click right-ctrl. Be aware that the mouse does not work properly if the QNX window is on the secondary screen.

After logging in to QNX, you should click “Utilities” and “Terminal”. Run the command “ifconfig” to see the ip address (you need it later) and then run the command “qconn”. You can then minimize or move the QNX away, as you don’t need to work on it directly.

2. Starting QNX Momentics IDE and Connecting to the Target

Open the QNX Momentics IDE program in Windows. You will find the program familiar, as it is a variation of Eclipse. As in Eclipse you can select the workspace location, and you can select what you want, but you must avoid having any spaces in the file path. You can remove the welcome screen, and click on “File->New->QNX C Project”. Select a name for your test program, and click next. In the project settings, you must select the “Build variants” panel, and select “X86 (Little Endian)”. Then you click “Finish”.

You will get a basic program that prints a welcome message. You need to build the program, by right clicking the project in the project explorer and select “Build Project”. Then you can run the program by right cliking the project and select “Run As->C/C++ QNX Application”. You will get the “Run Configuration” window. You need to add your target with the “Add New Target…” button. Insert the IP address of your target in the dialog box. Your configuration should then look like the image below. If your target is not connected, you have incorrect IP, or forgot you run “qconn” on the QNX terminal.



You can now click run, and the program will run. The printf text will be displayed on the Console within the program, even if the program actually runs on the QNX target. You only have to configure the target this first time. Be aware that you have to manually build the project each time you change it, it will not be built automatically when pressing the run button.

You can also use step-by-step debugging in Momentics as you have done in Eclipse.

When doing this exercise you will probably need to include the following headers:

```
#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <fcntl.h>
```

```
#include <sys/dispatch.h>

#include <sys/mman.h>
```

3. Shared memory

Shared memory is a special memory location that can be accessed by different processes. To use shared memory we need the following functions:

- shm_open – Opens a shared memory location.
- ftruncate – Sets the size of a file.
- mmap – map a memory region and return a pointer to it.

3.1. Creating shared memory

Here is a description of how a shared memory object is created.

1. Create the shared memory object with `shm_open()`. This function has three parameters:
    I. A location in the filesystem, like "`/sharedpid`". This will be found under `/dev/shmem` in the filesystem.
    II. Use O_RDWR | O_CREAT to specify that the memory should be read/write and it is being created.
    III. Use S_IRWXU.
2. The `shm_open()` returns a file descriptor, which is an integer.
3. Resize the shared memory location with `ftruncate()`. This function has two parameters:
    I. The file descriptor returned by `shm_open()`.
    II. The size of the memory object (`sizeof(<variable type>)`). You typically want to store a struct in the shared memory, so its size should be the size of the struct.
4. The shared memory must now be mapped into the address space of the process with `mmap()`. This function has six parameters:
    I. Should be 0.
    II. The size of the memory object.
    III. Use PROT_READ | PROT_WRITE to specify that the memory should be read/write.
    IV. Use MAP_SHARED to specify that it is shared memory.
    V. The file descriptor returned by `shm_open()`.

VI. Should be 0.

5. The `mmap()` returns a void pointer to the shared memory location, and data can be added to it. Remember to NOT change the pointer, as it will change what it points to. You must only change the data it points to.

> *When the pointer is a void pointer it means that it can point to anything. You can cast it to the pointer of the type you want, just remember to cast it to a variable with the correct size.*

3.2. Accessing shared memory

To access shared memory created by another process you follow a very similar method, except for the following:

1. You should not use the O_CREAT flag in the `shm_open()`.
2. You should skip the step with `ftruncate()`.

Remember to use the same location on the filesystem when running `shm_open()`.

Assignment A:

Create two programs (You could have two different projects in Momentics, but it is not strictly necessary). The first program should create a shared memory location, and store the struct described below. The struct contains the PID of this program, that you can get with the `getpid()` function.

```
struct pid_data{
   pthread_mutex_t pid_mutex;
   pid_t pid;
};
```

The struct contain a mutex, which should be used to protect the struct. To create a shared memory mutex, you should do the following commands, where ptr is the pointer returned by mmap() and myattr is a pthread_mutexattr_t.

```
    pthread_mutexattr_init(&myattr);
    pthread_mutexattr_setpshared(&myattr, PTHREAD_PROCESS_SHARED);
    pthread_mutex_init(&ptr->pid_mutex, &myattr );
```

The other program should read the pid from the shared memory, and print it. Verify that it is the correct value.

It is possible to run both programs from Momentics. With the triangle next to the green play button  you can select which of your previously started program to run. Each program will have its own console, and you can switch between them with this button:  You should also remember to stop any running program with the red stop button before starting new ones.

Alternatively you can copy one or both of the executables to the target and run them there. To copy files to the target, you should open the "Target File System Navigator" in Window->Show View->Other. It should be in the QNX Targets folder. With this view you can drag and drop files from the binary folder of your projects.

4. Messaging

Messages can be used for data transfer and synchronization between tasks. This is done using a client/server model. There are many functions for doing this, but we will stick to the six most basic ones:

- ChannelCreate – Run by the server to create a message channel.
- ConnectAttach – Run by the client to connect to a message channel created by the server.
- ConnectDetach – Run to disconnect from the message channel.
- MsgSend – Run by the client to send a message to the server. The function will wait until the server has replied to the message.
- MsgReceive – Run by the server when it wants to receive a message from the client. The function will wait until the server receives a message.
- MsgReply – Run by the server after it has received a message and performed the operations the client asked for. The reply can contain data to the client.

You can find more information about these functions either in the help section of Momentics or on the web:

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

QNX Momentics comes with some code samples and tutorials that you can use. Be aware that the IPC sample code used `name_attach()` and `name_open()` instead of `ChannelCreate()` and `ConnectAttach()`.

4.1. Message server

This is a description of a basic message server setup:

1. The server should first create a channel with `ChannelCreate()`. This function has one parameter:
    I. This parameter should be 0 for now.
2. The `ChannelCreate()` returns a channel id.

3. To wait for a message from the client, the server calls `MsgReceive()`. This function has four parameters:
    I. The channel id returned by `ChannelCreate()`.
    II. A pointer to a data buffer.
    III. Size of the buffer.
    IV. This parameter should be NULL for now.
4. When the server receives a message, the `MsgReceive()` function will return a receive message id, and the buffer will be filled with the data the client sent.
5. After the message is received, the server usually performs a job for the client.
6. When completing its job, the server call the `MsgReply()` function. This function has four parameters:
    I. The receive message id.
    II. A status value to the client, this is usually set to EOK (value = 0).
    III. A pointer to data sent back to the client.
    IV. The size of the data.

4.2. Message Client

This is a description of a basic message client setup:

1. The client should first connect to a channel with `ConnectAttach()`. This function has five parameters:
    I. Should be 0.
    II. The PID of the server process (which you should get from the shared memory you created in Assignment A).
    III. Should be 1.
    IV. Should be 0.
    V. Should be 0.
2. The `ConnectAttach()` returns a channel id.
3. Send a message to the server with `MsgSend()`. This function has five parameters:
    I. The channel id returned by `ConnectAttach()`.
    II. A pointer to the data you want to send to the server.
    III. The size of the sending data.

IV. A pointer to a data buffer for the data the server sends back.

V. Size of the data buffer.

4. This function will wait until the server have both received and replied to the message.
5. `MsgSend()` will return the status value the server provided in his `MsgReply()` function.
6. Data that the server returned will be found in the data buffer.
7. Run ConnectDetach() to disconnect from the server. The single parameter should be the channel id.

Assignment B:

Extend your program so the client sends a message to the server after getting its PID from the shared memory.

4.3. Multiple clients

A server often contains an infinite loop, where it waits for receiving a message, the replies it before waiting for the next. This allows one server to run continuously as several clients connect to it. If a client sends a message while the server performs an operation for another client, the message is queued.

When the server can get connections from several different clients, it is useful to know more about the message and the client that sent it. To do this, the server should provide a pointer to a `struct _msg_info` to the last parameter of `MsgReceive()`. This struct will then be filled with information about the message and its sender that can be used to print information about the client.

Assignment C:

Create a server that is able to handle several clients and messages. When receiving a message you should print a message telling the process id and thread id of the client, before replying.

The client application should start 4 threads that each sends a message to the server and prints the servers reply.

You can use the same POSIX thread functions as you did in normal Linux.

4.4. Thread priorities

In QNX you can give your threads different priorities, between the lowest priority 1 and the highest 63.

To set and get the priority of a thread, you can run the following functions from a thread.

```
int set_priority(int priority)

{

      int policy;

      struct sched_param param;
```

```
	// check priority in range
	if (priority < 1 || priority > 63) return -1;

	// set priority
	pthread_getschedparam(pthread_self(), &policy, &param);
	param.sched_priority = priority;
	return pthread_setschedparam(pthread_self(), policy, &param);
}

int get_priority()
{
	int policy;
	struct sched_param param;

	// get priority
	pthread_getschedparam(pthread_self(), &policy, &param);
	return param.sched_curpriority;
}
```

4.5. Priority Inheritance

We have already studied priority inheritance in an earlier exercise. In this exercise priority inheritance occurred when the priority of a low priority thread was raised because it had locked a resource a higher priority thread was waiting on.

Here we will look at another type of priority inheritance. Imagine that a server is running at a low priority, and a time consuming thread is running at a middle priority. If a high priority thread gets ready, it will preempt the middle priority, and start running. So far everything makes sense.

But then the high priority thread sends a message to the server, the server would then do some work before replying. Since the middle priority thread is higher priority than the server, the server will not be

able to do its job and replying the high priority job. This means that the high priority thread is prevented from doing its job by a thread with lower priority.

Priority inheritance is activated by default in QNX. This means that the server will inherit the client's priority when it receives a message from it. In the case described above, the server will run with the priority of the high priority thread.

You can disable priority inheritance of a message communication by using _NTO_CHF_FIXED_PRIORITY as a parameter to the `ChannelCreate()` function.

Assignement D:

Give the 4 threads from A different priorities. Give the server a priority so two threads have higher and two have lower than the server. The server should still print the process and thread ids of the client it receives a message from, so you know when it answers the different clients. Before and after each `MsgReceive()`, the server should print its current priority

Run the same program with and without the _NTO_CHF_FIXED_PRIORITY flag, and observe how it changes the priority of the server at different times.

> *You should give the main thread that creates the other threads the highest priority, so it will start all threads before any of the threads start running. If you don't do this, you might create one thread; let that run to completion before running the next one.*

5. System Profiler

The QNX target is able to log what is happening, and you can display this information in Momentics. Click the triangle next to this symbol  and select "Log Configurations". Right click "Kernel Event Trace" and select "New". Select target and save the log in the workspace of your message server. Go to the "Trace Settings" panel and set the "Period length" to 5. Save your log configuration and close the window.

Start the logging, and select "Always run in the background", this will prevent future loggings to show up as a window. You see that you get a trace file in your message server project. You can delete this file, as we need to run the logging while we are running the programs.

To log your programs, you need to first start the logging, run your server and then your client. You should be able to do this within 10 seconds. You should then double click the trace file that show up in your server project. Say yes to open the "QNX System Profiler" perspective and if you have to repair the log. You will see a summary page that shows different statistics from your logging.

Click "System Profiler->Display->Switch Pane->Timeline". You will now see a list of the activities of all the processes during the 5 seconds. We are only interested in our two processes, so right click and select "Filters…" You should get a window of all the processes, and deselect all except yours. Click the "+" next to your processes in the timeline to show the threads.

You can see the whole 10 seconds of the logging, and the activity of your program is very short. Mark a period in the timeline and zoom in on where your processes perform with the zoom tools. You have a list of events in the "Trace Event Log" below the timeline. You can see IPC events in the timeline by clicking the  button.

- Green color is a running task
- Dark green is a suspended task
- Red is a task sending a message
- Blue is a task receiving a message

Use the timeline to understand how your programs execute. The "Trace Event Log"-tab shows what type of action is performed and some additional information, like senders and recipients in message passing.

To get back to c/c++ perspective you can either click the c/c++ perspective icon in the top right corner, or "Window->Close perspective".

Since we are running in a virtual machine, the response times and time to perform different operations will vary. The order of execution will be correct though.

6. Approving the exercise

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

1. You only need to show a working program from Assignment D, as it covers everything that has been done before.
    - How are the priority of the server different when using _NTO_CHF_FIXED_PRIORITY and not?
    - Show where the shared memory is located in the file system.
2. Show the timeline and explain what is happening.