Exercise 9: QNX IPC

In this exercise we will use the QNX Neutrino RTOS. This is a microkernel OS, where only the scheduler, timers, interrupt handling and networking is run in the kernel, and everything else is run as user-space processes. In order to tie these processes together, the kernel uses a message-passing system. This inter-process communication will be the focus of this exercise.

We will be running QNX on a virtual machine, and use the QNX Momentics IDE to cross-compile from Windows to QNX.

1. Getting started

1.1. Setting up QNX on a virtual machine

Download the file "QNX Target.ova" from Blackboard, and open it with VirtualBox (double click). In the import dialog, scroll down to "USB Controller" and disable it. Then click "Import" - this will take a few seconds. You can now start the QNX target by double clicking on it, or selecting "Start" from the top ribbon menu.

The virtual machine window will capture your mouse and keyboard input, which means that in order to move your mouse cursor back to the host machine, you have to input a special command. This command is shown in the bottom right of the VirtualBox window, and is by default a single click of the Right Control key.

When booting, either let the options time out, or select the defaults. You will then be greeted with a login screen. Type "root" as the user, and leave the password blank.

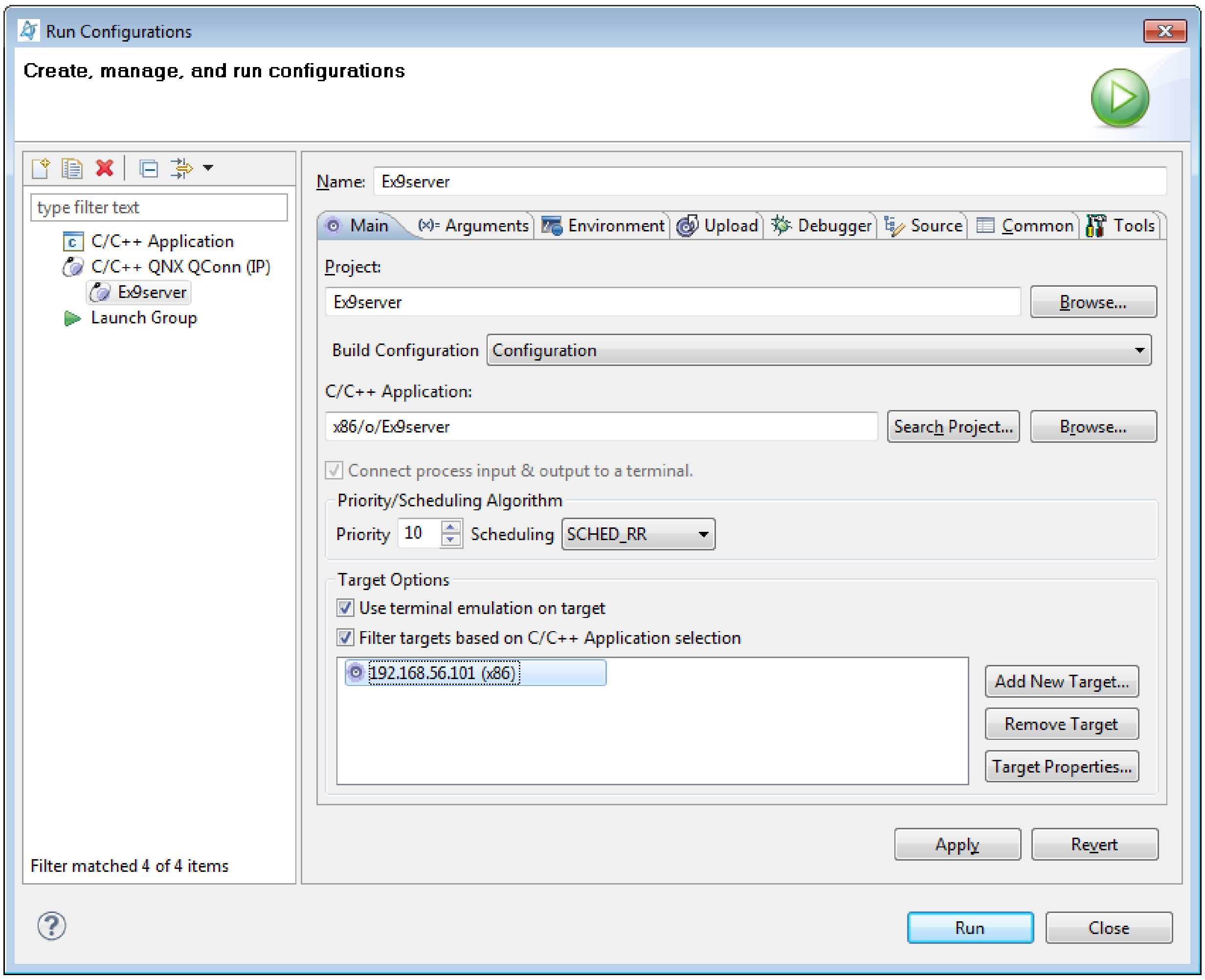
In order to cross-compile from the QNX Momentics IDE, we need to know the IP address of the virtual machine, and also enable the remote IDE connection. Run `ifconfig` to show the IP address, then run `qconn`. From now on, we don't really need to work on the QNX machine directly.

1.2. Connecting the QNX Momentics IDE to the target

Open the QNX Momentics IDE, and select a directory for your workspace (note that the workspace path cannot have any spaces in it). Then open "File > New > QNX C Project". Choose a name for your project, and click "Next". In the project settings, go to the "Build variants" panel, and select "x86 (Little Endian)", both debug and release. Click "Finish". You may have to click "go to the workbench", the rightmost circular icon, then the editor will show up.

You will get a default program that prints a welcome message. Build the program by right-clicking the project in the project explorer, and selecting "Build Project" (or by clicking Ctrl+B). You can then run the program by right-clicking the project and selecting "Run As > C/C++ QNX Application" (or Ctrl+F11).

The "Run Configuration" window should pop up. Click "Add New Target...", and enter the IP address you found previously. Your configuration should then look like shown in the image below:


Run Configurations

Create, manage, and run configurations

Name: Ex9server

Project: Ex9server

Build Configuration: Configuration

C/C++ Application: x86/o/Ex9server

[✓] Connect process input & output to a terminal.

Priority/Scheduling Algorithm

Priority 10 Scheduling SCHED_RR

Target Options

[✓] Use terminal emulation on target

[✓] Filter targets based on C/C++ Application selection

192.168.56.101 (x86)

Filter matched 4 of 4 items

You can now click "Run". The `printf` output will be shown in the terminal window in the IDE, even though the program is being run on the QNX machine.

You only have to perform this configuration once. But be aware that you will have to manually build the project each time you make changes to it, as only clicking the Run button will not also build it.

2. Channels and message passing

2.1. Message passing API

Message passing is used between all QNX processes, either directly in the application code, or as part of the underlying mechanisms for implementing the POSIX interface. In this exercise, we will be using the message passing primitives directly.

Messages are not sent between processes directly, but over channels. These channels must be created on one end, and attached to on the other end - which means that one process must be responsible for creating the channel, while the other process must attach itself to the channel at a later point. This effectively imposes a client-server model to the structure of our process hierarchy.

- **ChannelCreate**`(int flags)`
  Run by the server

flags:	0 by default, see docs for more

- **ConnectAttach**`(int nd, int pid, int chid, int index, int flags)`
  Run by one or more clients

nd:	Node descriptor, `ND_LOCAL_NODE` (or just 0) for this machine
pid:	Process ID of the channel creator
chid:	Channel ID of the created channel
index:	Lowest acceptable connection ID - set this to 0
flags:	Set this to 0, see docs for more

- **ConnectDetach**`(int coid)`
  Run by the client to disconnect from the channel

coid:	The connection ID to break

- **MsgSend**`(int chid, void* smsg, int sbytes, void* rmsg, int rbytes)`
  Run by the client to send a message to the server
  This function will wait until the server sends a reply

chid:	ID of the channel
smsg:	Pointer to buffer containing the message
sbytes:	Size of the message being sent
rmsg:	Pointer to buffer where the reply can be stored
rbytes:	Size of the receive buffer

- **MsgReceive**`(int chid, void* msg, int bytes, struct _msg_info* info)`
  Run by the server to receive messages
  Returns a receive identifier (int), which is used for MsgReply

chid:	ID of the channel
msg:	Pointer to buffer where the message can be stored
bytes:	Size of the receive buffer
info:	NULL, or pointer to `_msg_info` struct to store additional info

- **MsgReply**`(int rcvid, int status, void* msg, int bytes)`
  Run by the server after it has received and handled a message
  The reply can contain data to send back to the client

rcvid:	The receive ID previously returned by MsgReceive
status:	The status to unblock the sending client, typically 0 (EOK)
msg:	Pointer to buffer containing reply to client, or NULL for no message
size:	Size of the buffer

These functions are found in `sys/dispatch.h`

You can find more information about QNX message passing here:
http://www.qnx.com/developers/docs/6.5.0_sp1/topic/com.qnx.doc.neutrino_sys_arch/ipc.html

2.2. Announcing the server process and channel IDs

In order for the client to connect to a channel, it needs to know the process ID of the server process, as well as the channel IDs of any channel the server has created. Obviously we can't communicate this information by using channels and message passing, so we'll have to use some other option.

The simplest option is to create a file in some known location in the filesystem, to which the server writes its information once it has been started. Any clients then look for this file, then read the process ID and channel ID, then connect to the server.

For programs where the client and server are part of the same process (but different threads), the process ID is known by all threads, and the channel ID can be passed as an argument to the call to `pthread_create`.

Task A:

Create a client and server that performs a single message passing operation. The client and server can either be separate processes, or threads within a single process.

The server should create the channel, and announce its process and channel ID to any clients. You can get the process ID with `getpid`, from `unistd.h`. The server should then enter an infinite loop waiting for messages, then replying to them as they come in.

The client should acquire the process and channel ID, connect to the channel, send a message and await its reply, then disconnect from the channel.

Instrument your code with enough `printf`'s to make sure that this is working as you would expect.

If you are creating multiple processes, you can run both programs from the IDE simultaneously by clicking the dropdown arrow next to the green play button in the lower panel, or by pressing Ctrl+F11 for each project you wish to run. You can also switch terminal windows by clicking the blue monitor dropdown, on the top right ribbon of the console window at the bottom. Remember to stop all running programs with the red stop button before starting new ones, and also close all windows for terminated programs with the grey X or double Xx.

Open, write, read, and close files with `fopen`*,* `fprintf`*,* `fscanf`*, and* `fclose`*. You can store the file in something like* `/root/serverinfo.txt`*. Note that you have to close the file for your data to be stored.*

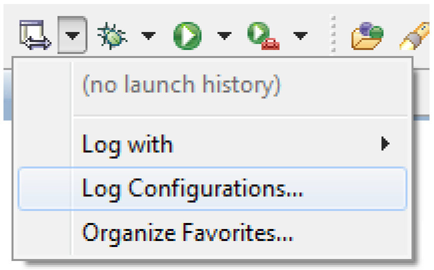
Task B:

Connect four clients to the server. Get the sender's process and thread ID by passing an empty `_msg_info` struct to the call to `MsgReceive`, and print them out when receiving the message.

If you have split the client and server into separate processes, you should probably spawn multiple threads in the client process (instead of starting lots of processes) - just because it's easier to deal with in the IDE.

2.3. *Priority inversion and inheritance*

All QNX threads have a priority and a scheduling policy. The priority ranges from 1 (lowest) to 255 (highest), independent of the scheduling policy. Unprivileged threads can only have a priority between 1 and 63. The scheduling policies are SCHED_FIFO, SCHED_RR, and SCHED_SPORADIC, the default being round-robin.



Functions for getting and setting the thread priority are provided on Blackboard.

We have already studied priority inheritance in the Xenomai Semaphores exercise, where the priority of a low priority thread was increased because a higher priority thread was waiting on their shared resource. Here we will look at priority inheritance again, but in a message passing context.

Imagine a low-priority server being accessed by a high-priority client. If there is some time-consuming middle priority process running, it will preempt the server and prevent it from responding. As before, this effectively "inverts" the priority hierarchy, letting the middle priority thread run while the high priority thread waits.

Priority inheritance is enabled by default in QNX. This means that a server will inherit the priority of the client it receives a message from. You can disable priority inheritance on messages sent over a channel by passing `_NTO_CHF_FIXED_PRIORITY` as a flag to `ChannelCreate`.

Task C:

Give the four client threads and the server different priorities, such that two client threads have a higher priority than the server, and two have a lower one. Make sure you know which thread ID has which priority level (either have the client print this out, or get it from the `_msg_info` struct). Before and after a call to `MsgReceive`, the server should print out its own priority.

Run the same program with and without the fixed priority flag, and observe how the behavior differs.

You should give the main thread the highest priority, so that it starts all threads before any of them preempt the main thread and prevent it from spawning the rest of the threads.

3. Using the system profiler

QNX has extensive logging capability, which Momentics can display in a timeline. Click the QNX Logging button (see screenshot), and select "Log Configurations". Right click "Kernel Event Trace", and select "New". Select the target, and save the log in the workspace of the server. Go to the "Trace Settings" tab, and set the "Period Length" to 5 seconds. Click "Apply".

Start the logging by clicking "Log", and select "Always run in the background". This will prevent future logging to pop up as a window. You can see that you get a trace file in your project. You can delete this file, as we need to run the logging while we are running our program(s).

To log your programs, you need to first start the logging, then run the server and clients. To start the logging, click the same dropdown menu as before, and select the log configuration that shows up at the top (it should replace the "no launch history" text in the screenshot above). After 5 seconds have passed (or whatever you set the period length to), the logging will stop. Then double click on the trace file that shows up in the project. Say "Yes" to open the QNX System Profiler perspective, and also say "yes" if you have to repair the log. You should see a summary page that shows different statistics from your logging.

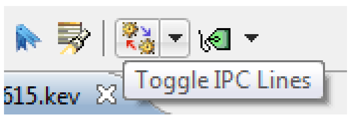
From the top menu, click "System Profiler > Display > Switch Pane > Timeline". You will now see a list of all the activities that were logged during the 5 seconds. We are only interested in our own processes, so right click on the timeline and select "Filters...", and deselect all except your own. Click the "+" next to your processes in the timeline to show the threads.

The activity of your program is very short, so mark a period in the timeline and zoom in on where your processes are running. You have a list of events in the "Trace Event Log" below the timeline. You can see the IPC events in the timeline by clicking the "Toggle IPC Lines" button.

• Bright green:	Running
• Dark green:	Suspended
• Red:	Blocked on sending
• Blue:	Blocked on receiving

Use the timeline to understand how your programs execute, and compare with and without priority inheritance.

To get back to the C/C++ perspective (to get back to the editor), either click the C/C++ button in the top right corner, or click "Window > Close perspective".


615.kev
Toggle IPC Lines