Debugging the Kernel with VMWare#

Hopefully you here via the Debugging The Kernel page. If not, I'd highly recommend you start there, since it covers the basics of debugging the kernel.

This page describes the specifics of debugging a QNX Kernel that is running on a VMWare virtual machine. This page was originally written to be windows specific; where the steps between windows and linux differ, a section has been added for the linux instructions.

VMWare Workstation#

VMWare Workstation (the latest version is 6.0) is a useful tool in that it allows you to run one or more virtual x86 cpus on your development system. As far as the OS is concerned, it's an independant x86 machine, complete with BIOS, serial, disk and network devices.

Our standard qnxbasedma.build and bios.build will work with vmware.

Step 1: Build a bootable image#

Given that your kernel source environment is not on the vm, you need some way of getting the vm to boot your kernel. VMWare supports PXE netbooting, but probably the simplest method to load a fresh image is to generate a boot floppy image. You can do this with the mkifs filter, mkifsf_vmware.

To generate a floppy boot image with this filter, simple replace

[virtual=x86,bios +compress]

with

[virtual=x86,vmware +compress]

NB: If you see it complaining about not being able to find a watcom ld when you build your image, then you need to need to update to QNX 6.3.2 or later...

Step 2: Download and install the vmwaregateway application

VMWare can redirect it's serial port to a named pipe. gdb doesn't support connected to windows named pipes (why not?!!) but there is a useful util called vmwaregateway.exe that is over on the L4Ka site. You can get it here:

http://l4ka.org/tools/vmwaregateway.php

Install it, register it as a services and start the service.

```
vmwaregateway.exe /r
```

followed by

```
net start vmwaregateway
```

Under Linux#

Under Linux, we use a Unix Domain Socket instead of a named pipe. gdb doesn't support direct connections to Unix Domain Sockets. We can, however, use a program called socat (one particular evolution of netcat) to create a tcp interface for the socket.

Step 3: Add a serial port to your vmware configuration#

Go to the add hardware wizard, select a serial port and then redirect the output to the named pipe `\\.\pipe\vmwaredebug`

In the settings you should specify "This end is the client" "The other end is an application" and "Yield CPU on poll"

Under Linux#

We need to set up two serial ports. The reason for this is that if we try to run gdb on the first serial port and we get a kprintf over that serial port, then our gdb session may become corrupted. In order to keep things simple, we'll create the first serial port to output to a regular file, this way we can keep track of the traffic on that serial port by simply catting or tailing the file. The second serial port, on the other hand, needs to be a socket so that we can send two-way traffic over it. When selecting the output for the second serial port use the "Output to socket" option. For additional simplicity, lets name our serial ports like so, to keep them in order:

/tmp/vmwaredebug0
/tmp/vmwaredebug1

In the settings for our second serial port, we need to specify that "This end is the server" and "The other end is an application", as well as selecting the option to "Yield CPU on poll"

Step 4: Add a floppy image to your vmware configuration#

Go the add hardware wizard, select floppy drive and specify the boot image you created in your build file

Step 5: Make sure that your debug port goes to serial - not bios!#

startup-bios -D 8250.2f8.38400 -K 8250.3f8.38400 -v
gdb_kdebug -K -D1

Your debug port is now going out the second serial port, even though you probably don't have one setup. You need this though since otherwise when you 'ungrab' the input from vmware with ctrl-alt, you will abort the kernel debugger!

Step 8: connect with gdb#

Start the vmware session. If you specified -K to gdb_kdebug then it should hang waiting for gdb to connect.

kdebug uses the gdb remote protocol. So load the procnto-instr.sym into gdb, and then connect to localhost port 567

(gdb) file procnto-instr.sym
(gdb) target remote localhost:567

You should now be live! Set a breakpoint and hit the slopes, dude!

Under Linux#

Once you start your vmware session, vmware will create your Unix Domain Sockets. Now it's time to put socat to work.

socat -d -d /tmp/vmwaredebug1 TCP4-LISTEN:65333

We use the second serial port because it is the one we are running the kernel debugger on. If you want to connect your first serial port, use `/tmp/vmwaredebug0` instead of `/tmp/vmwaredebug1`, and use a different port for TCP4-LISTEN for each serial port you have connected. now we can connect to kdebug using the remote protocol:

(gdb) file procnto-instr.sym
(gdb) target remote localhost:65333

... and now, just like windows, you're running live!

Lastly...#

A couple of gotchas

- Our mix of cygwin and non cygwin tools don't play well with each other. If you have cygwin installed, don't BUILD from a cygwin mountpoint, and make sure you pass windows pathnames (but strangely, with unix forward-slashes) for command line arguments. Yay.

Under Linux#

every time your virtual machine restarts, the sockets used for the serial port output go away, so when the machine comes back up socat will need to be run again if you want to reconnect to the serial port.