Using IDA's GDB debugger with QEMU emulator

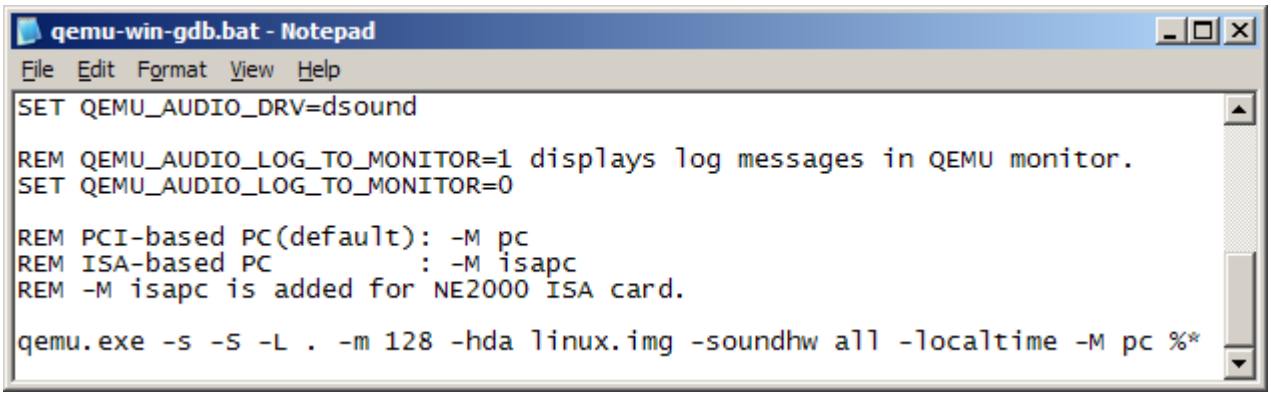
QEMU is a processor emulator which can emulate a handful of processors, including Intel x86 and ARM architectures. It includes a GDB stub which can be used with new GDB debugger plugin in IDA 5.4.

Getting QEMU

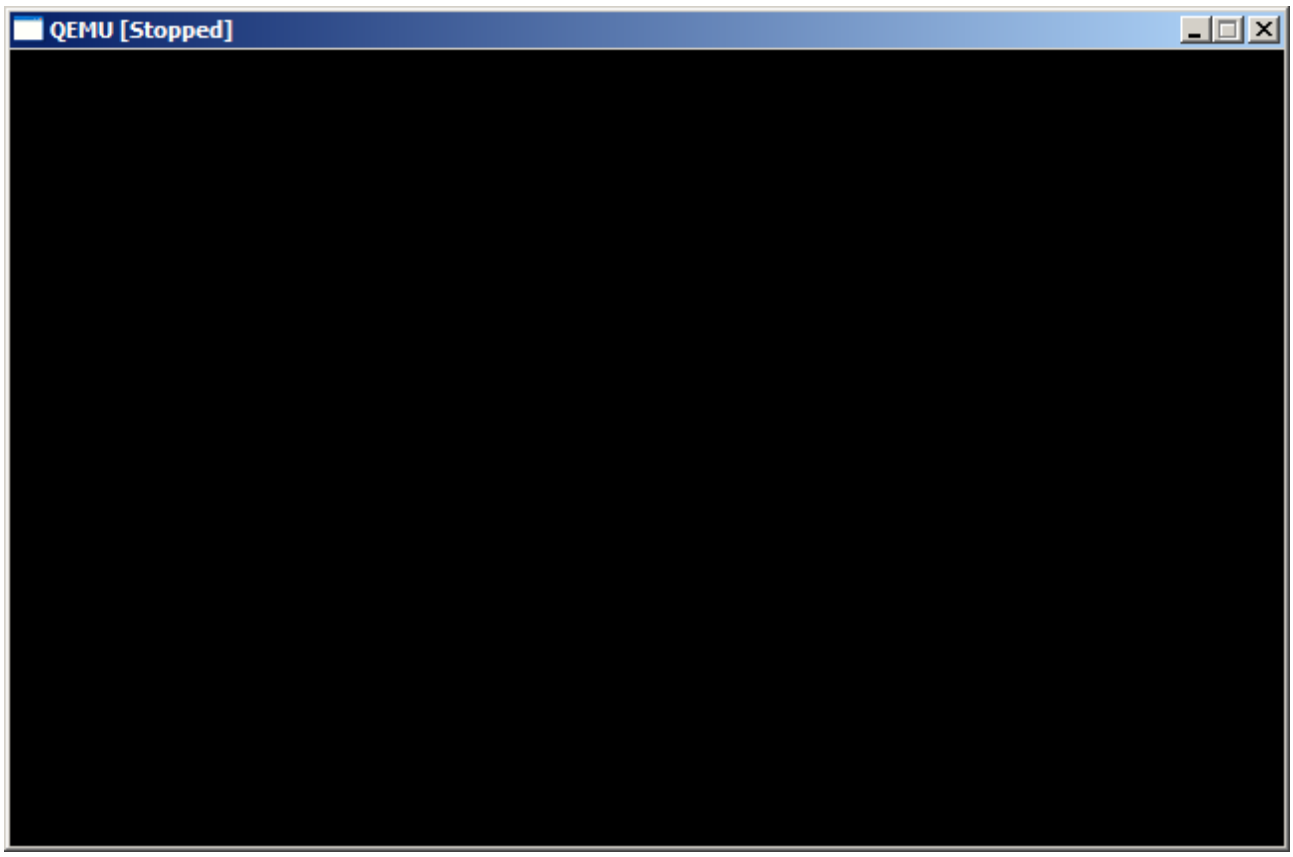
QEMU's home page is at http://bellard.org/qemu/. Win32 builds can be downloaded from Takeda Toshiya's page at http://homepage3.nifty.com/takeda-toshiya/qemu/. This primer assumes you downloaded QEMU 0.9.1 for Win32 or later. We will debug the small Linux included with QEMU.

Enabling GDB stub

After unpacking QEMU, make a copy of the `qemu-win.bat` file, for example `qemu-win-gdb.bat` and edit it. Add `-s -S` to the qemu.exe call (-s enables GDB stub and -S instructs QEMU to stop at the system start):

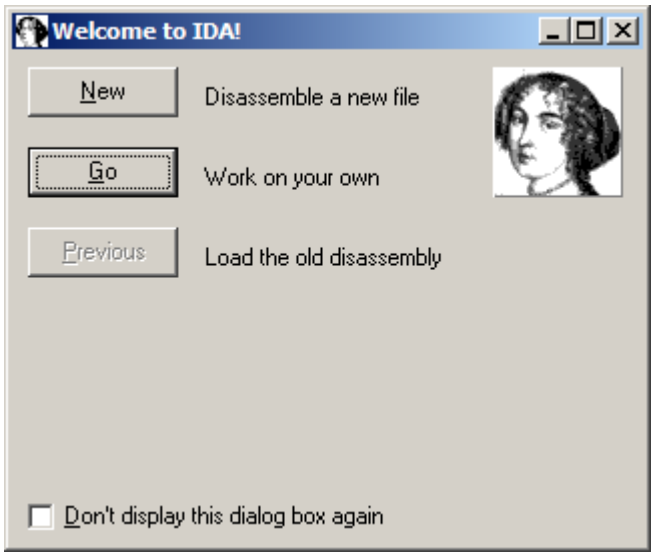


Run the .bat file. QEMU will stop and wait for the debugger.

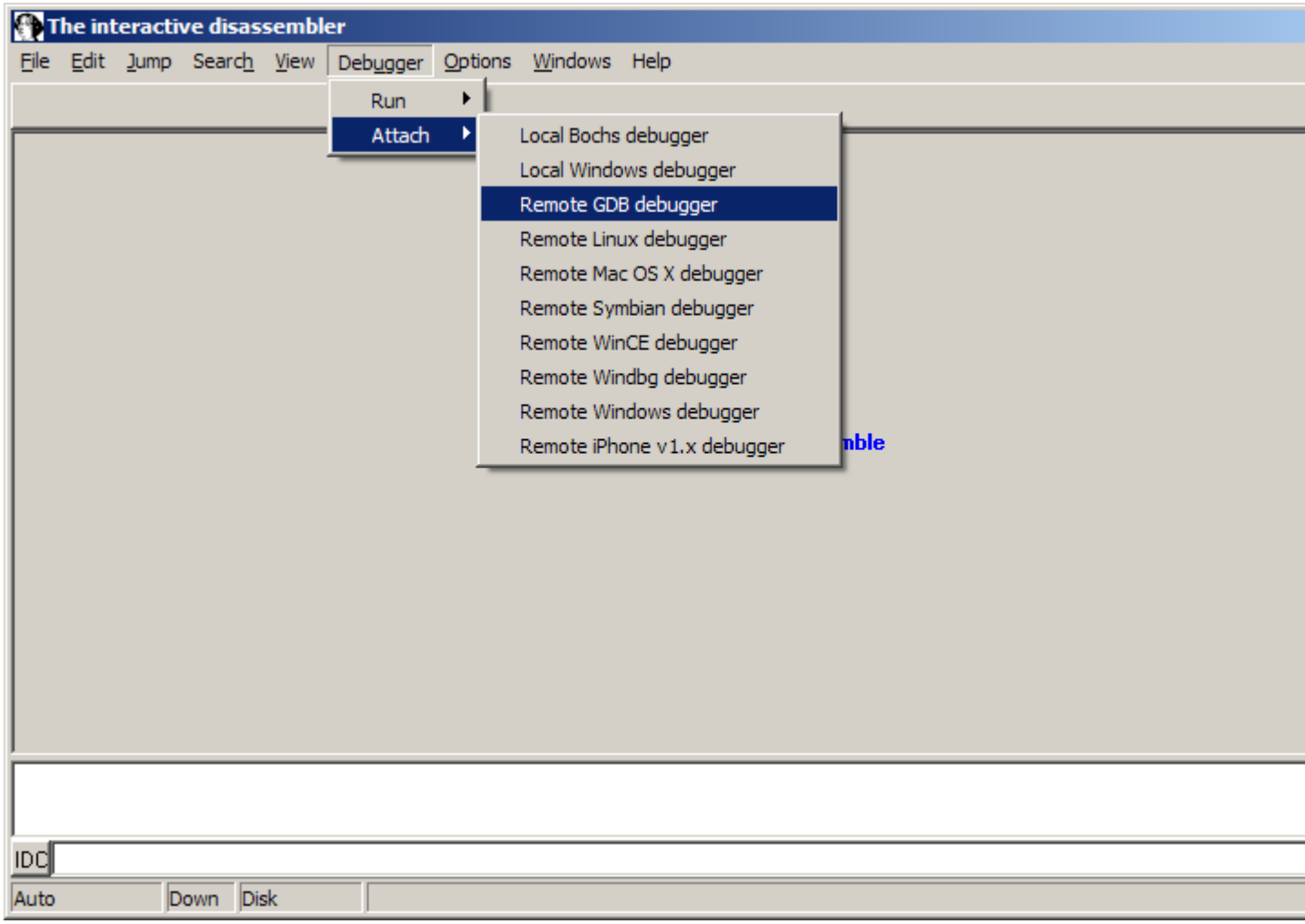


Debugging with IDA

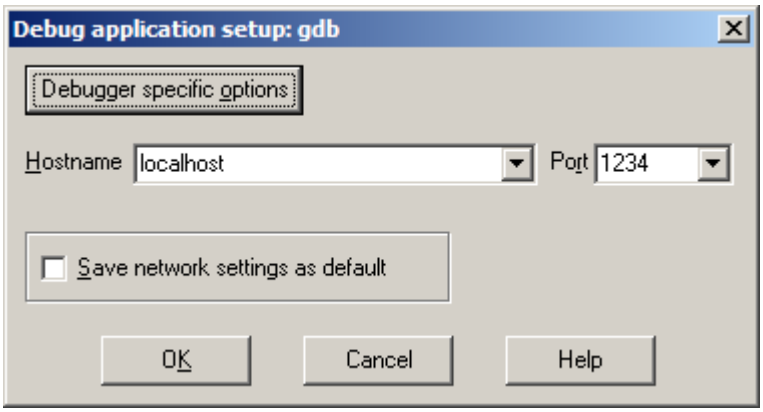
Start IDA.



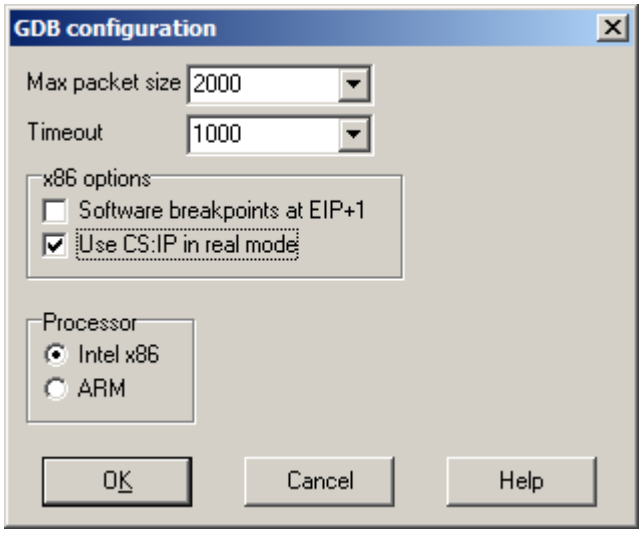
If you get the welcome dialog, choose "Go".



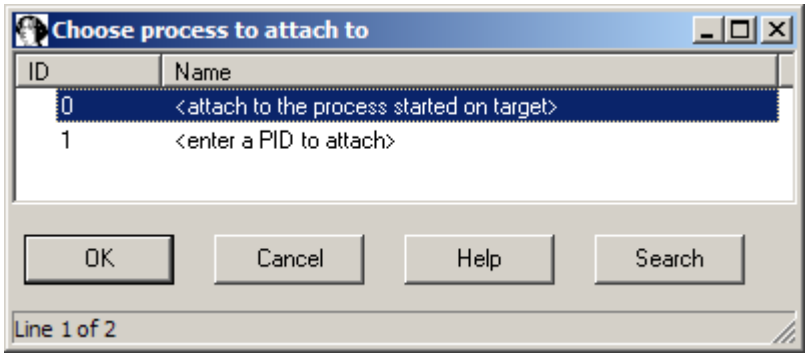
Choose Debugger | Attach | Remote GDB debugger.



Enter "localhost" for hostname and 1234 for the port number. Click "Debugger specific options".

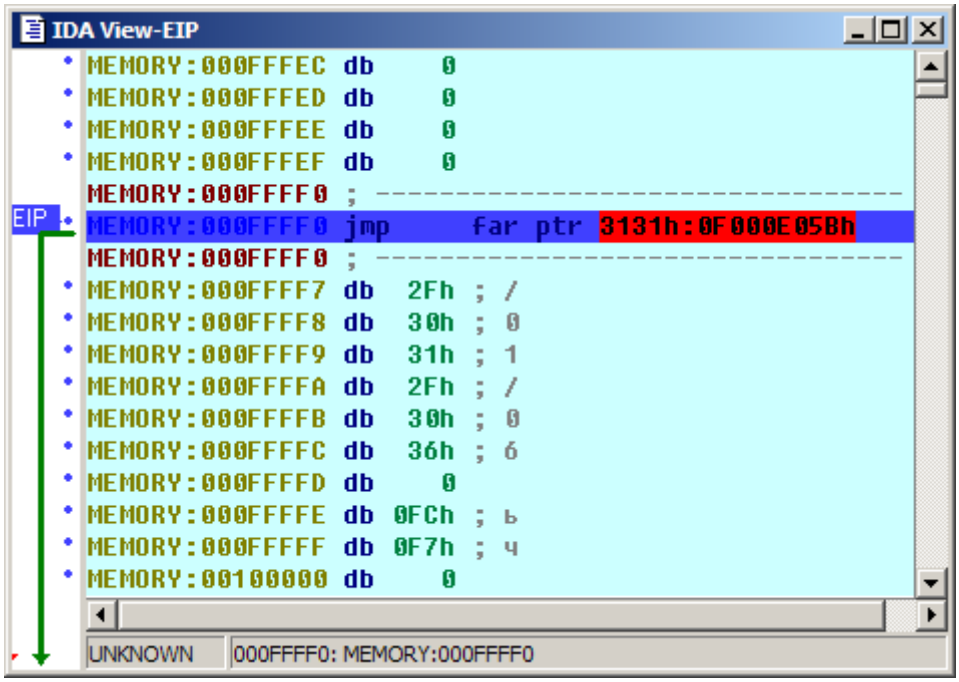


QEMU needs special configuration because it behave slightly differently from other GDB stubs. Uncheck "Software breakpoints at EIP+1" and check "Use CS:IP in real mode". Make sure Processor is set to "Intel x86". Click OK, then click OK in "Debug application setup" dialog.



Choose <attach to the process started on target> and click OK.

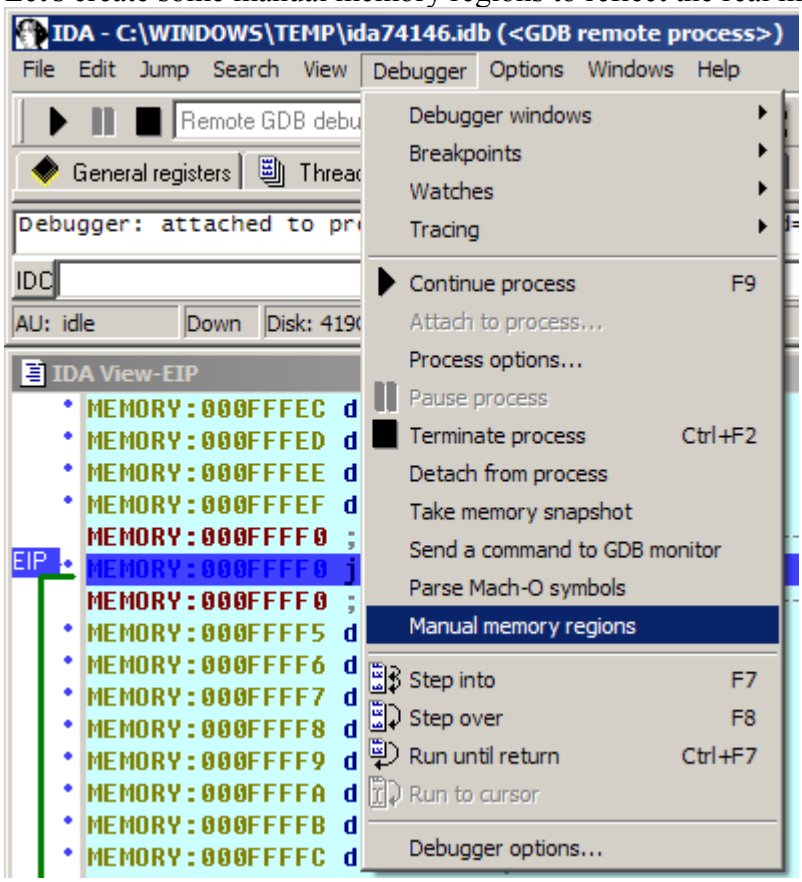
Debugging the BIOS



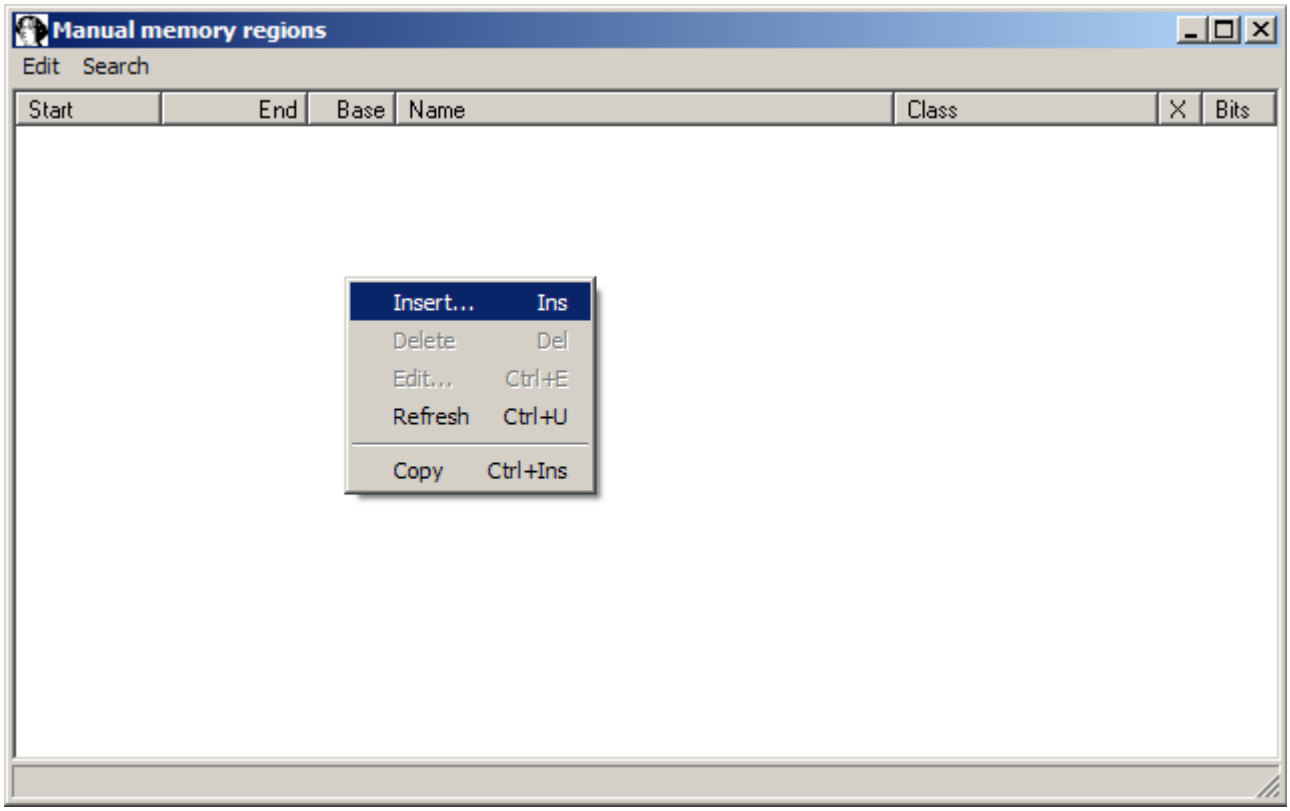
The BIOS entrypoint is displayed but the disassembly is wrong since the default MEMORY

segment is 32-bit.

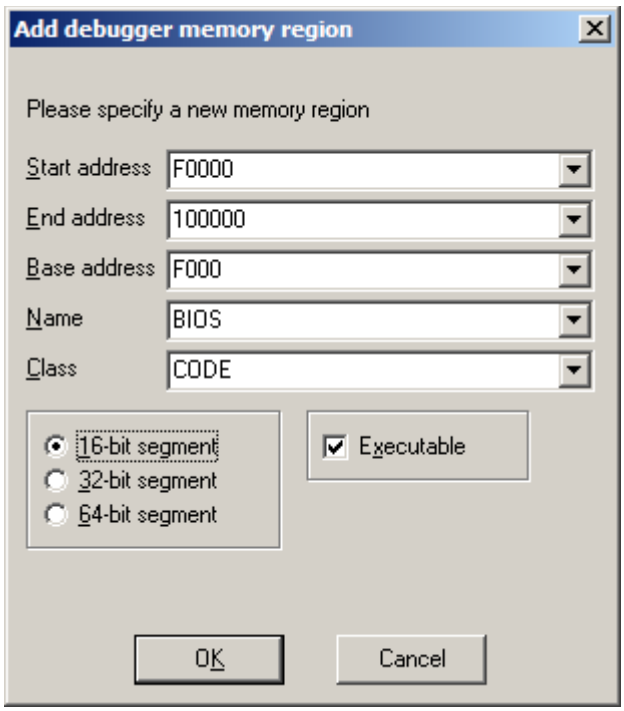
Let's create some manual memory regions to reflect the real memory map.



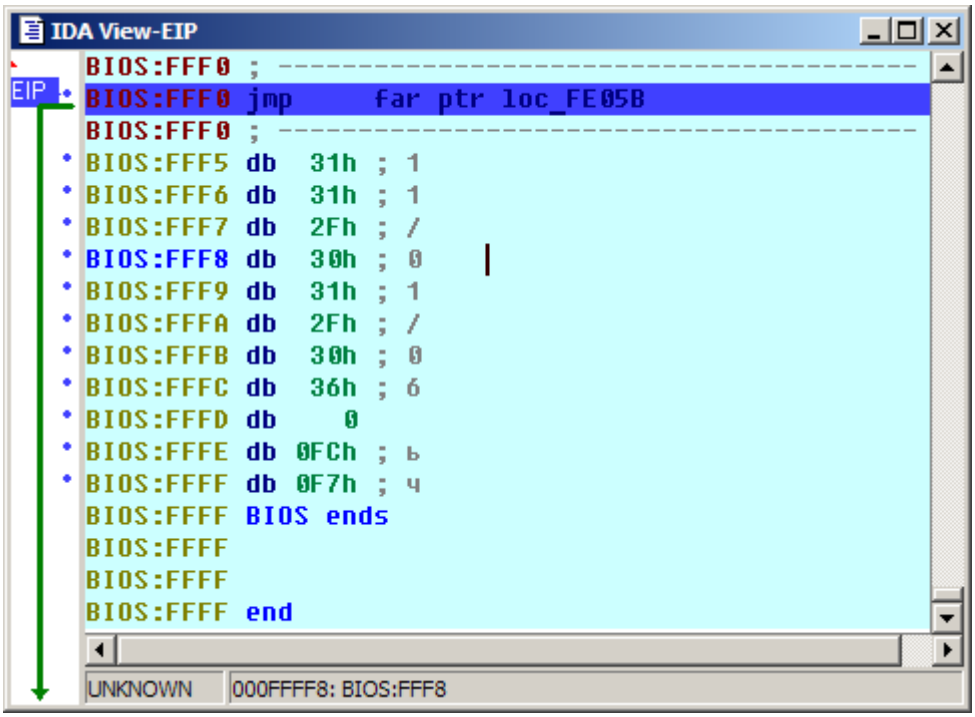
Go to Debugger | Manual memory regions.



Right-click and choose "Insert..." (or press Ins).



Enter the following details:
Start address: F0000
End address: 100000
Base address: F000
Name: BIOS
Class: CODE
select "16-bit segment"
Click OK.

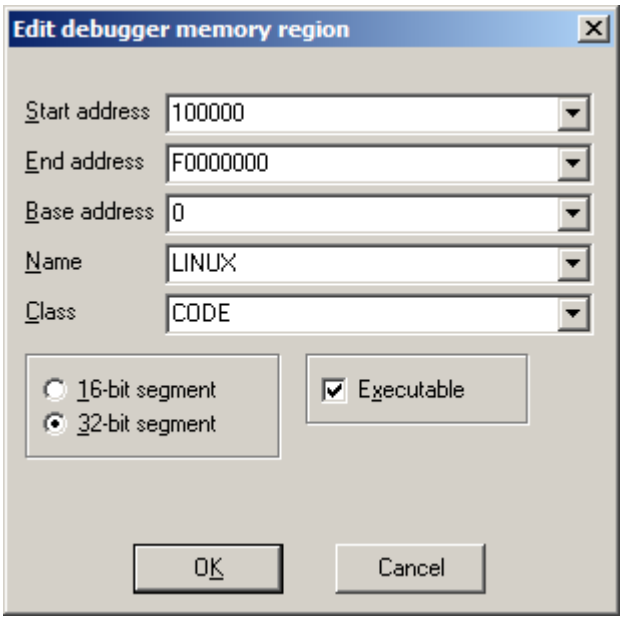


Now the disassembly is correct and you can trace the BIOS code.

Debugging the kernel

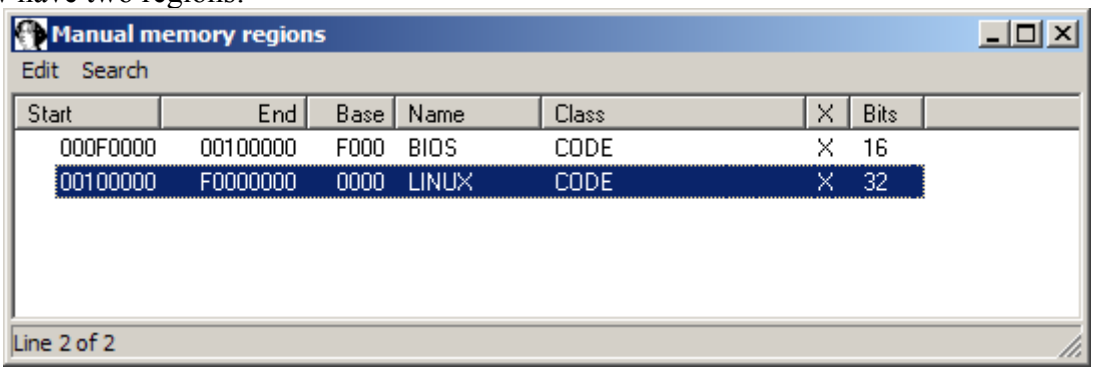
To debug the Linux kernel, we first need to create memory regions where it will execute. The usual kernel entrypoint is at address 100000, and it can use memory almost to the maximum 4G address.

Open memory regions list (Debugger | Manual memory regions) and add a new region:

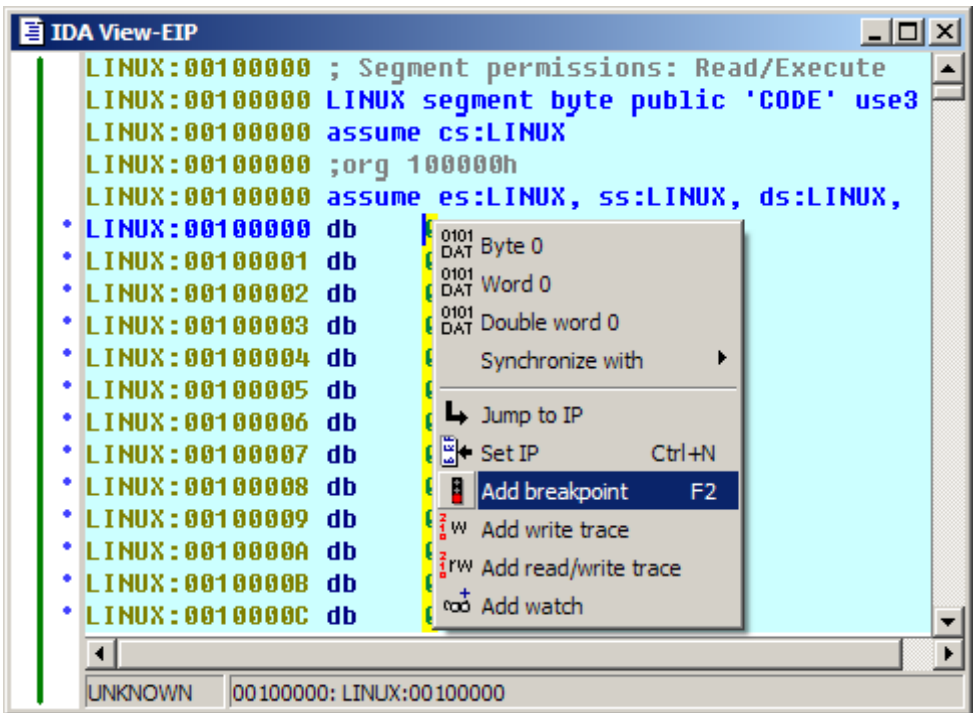


Start address: 100000
End address: F0000000
Base address: 0
Name: LINUX
Class: CODE
select "32-bit segment"
Click OK.

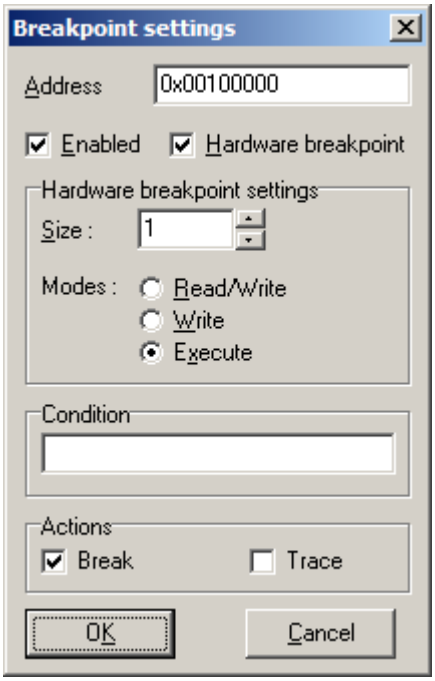
We now have two regions:



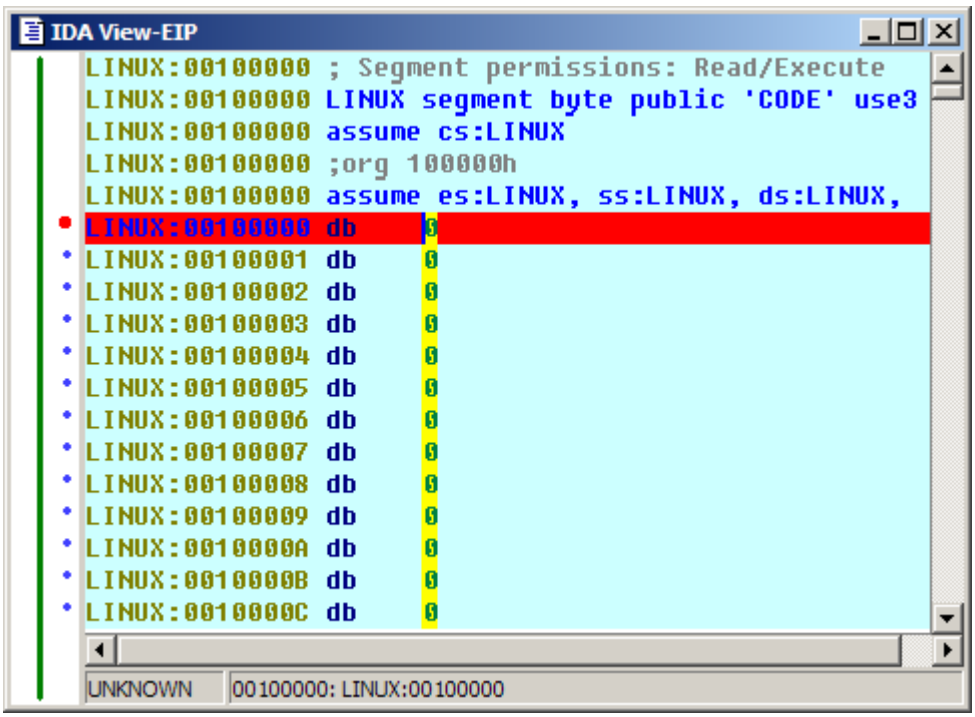
Double-click the LINUX region to go to its beginning.



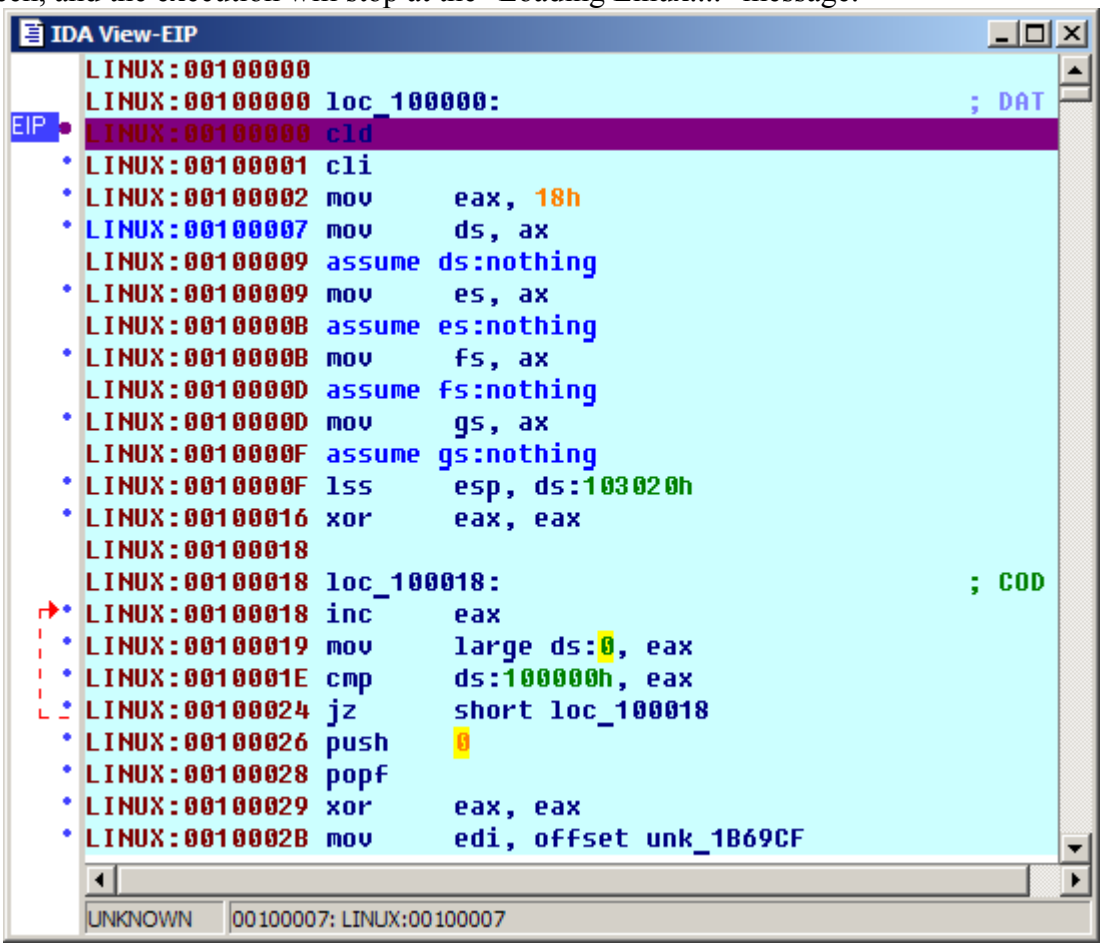
Press F2 or choose "Add breakpoint" from the context menu.



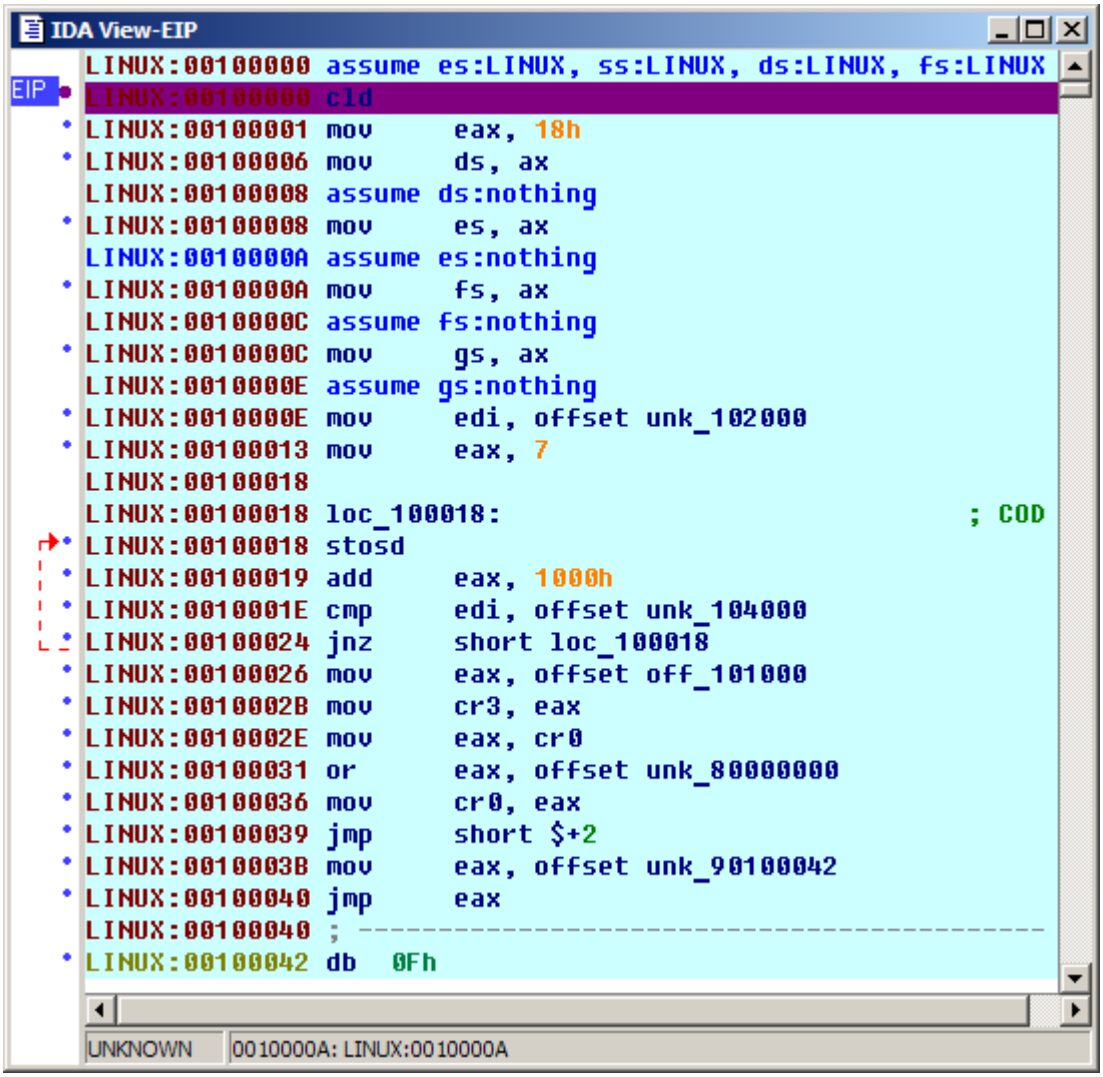
Check "Hardware breakpoint" and select "Execute" in "Modes". Click OK.



Now press F9 or choose Debugger | Continue process. You should see BIOS and LILO messages on the screen, and the execution will stop at the "Loading Linux...." message.



This is the initial loader which decompresses the kernel. If you press F9 once more, you'll see "Uncompressing Linux..." and then "Ok, booting the kernel" messages in QEMU and IDA will stop at the decompressed kernel entrypoint:



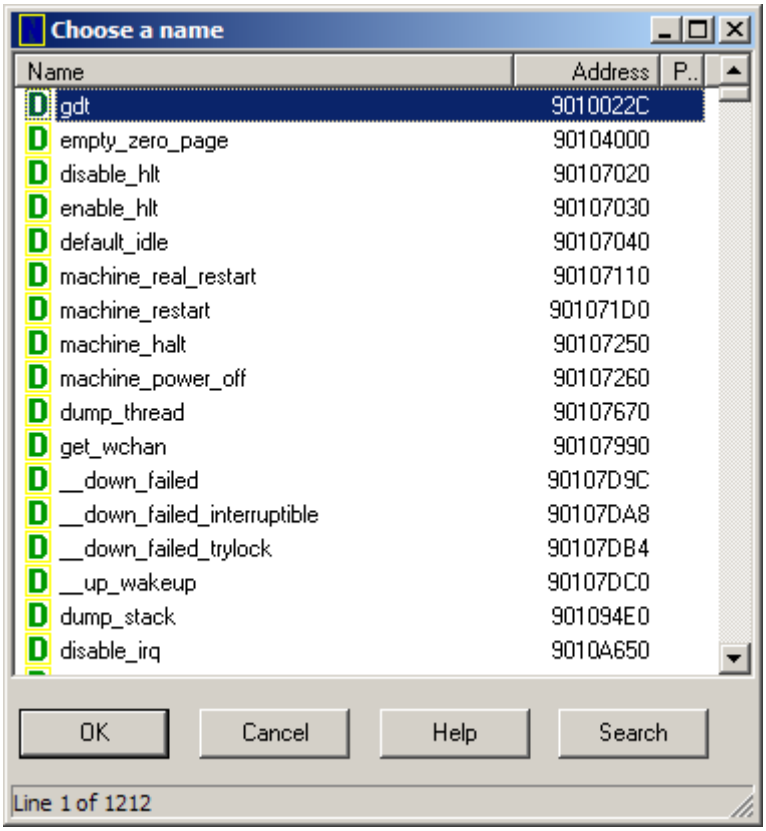
This code sets up paging table, enables paging, and then jumps to the "real" kernel entrypoint.

Adding symbols

Kernel symbols are available as `/proc/ksyms` or `/proc/kallsyms` pseudo-file after booting. If you get that file from the VM to the host, you can add the symbols to your disassembly. Go to File | Python command... and enter the following short script:

```
ksyms = open(r"D:\ksyms") #path to the ksyms file
for line in ksyms:
  addr = int(line[:8], 16)
  name = line[9:-1] # use line[11:-1] in case of kallsyms
  idaapi.set_debug_name(addr, name)
  MakeNameEx(addr, name, SN_NOWARN)
  Message("%08X: %s\n"%(addr, name))
```

Click OK and wait a bit until it finishes. After that you should see the symbols in the dissassembly and name list:



Happy debugging!