Debugging The Kernel#

Ok, you've built your kernel, booted it, and now it's crashed! What's going on?

Getting Started#

In order to use the kernel debugger, you first need to install the gdb_kdebug bootstrap executable into your boot image.

If you haven't already, checkout and do a make install of lib/kdutil and services/kdebug The resulting binary is called gdb_kdebug.

You can then add gdb_kdebug to your build file, in the boot section, like so...

```
[virtual=x86,bios +compress] boot = {
   # Reserve 64k of video memory to handle multiple video cards
   startup-bios -s64k -Dconsole -K3f8.9600

   gdb_kdebug

   # PATH is the *safe* path for executables (confstr(_CS_PATH...))
   # LD_LIBRARY_PATH is the *safe* path for libraries (confstr(_CS_LIBPATH))
   #    i.e. This is the path searched for libs in setuid/setgid executables.
[+keeplinked]
   PATH=/proc/boot:/bin:/usr/bin:/opt/bin LD_LIBRARY_PATH=/proc/boot:/lib:/usr/lib:/lib/dll:/opt/lib procnto-instr -P
}
```

The [+keeplinked] generates a relocated procnto-instr.sym binary that you can use as a symbol file for gdb (see below)

Kernel Debugger Command-line Options#

-a	Disables asynchronous check for break detect
-v	Make more verbose
-K	Set inital breakpoint at kernel entry
-U	Add user signals to set of interesting faults
-P[0|1]	Select standard remote protocol or acme protocol
-D[0-9]	Select debug device <n>

Startup Debug Devices#

The kernel debugger makes use of the debug device callouts that were recorded in the system page during startup. These are normally specified with the -D and -K options to startup (see startup-bios for details). You can select which one of these is used as the kernel debugger device by using the -D option, followed by the number of device to use (starting at zero).

Note that if you add the -v option, then the debugging output is always sent to device zero, so if you are already using that for the kernel debug device then things probably won't work.

Also note that you shouldn't be running a normal device driver on it, since it will interact in some painful ways with your kernel debugging. You may need to hit /etc/system/enum/devices/char to remove the automatic starting of devc-ser8250 at bootup time.

Entering the Kernel Debugger#

There are a couple of ways of entering the kernel debugger. The most obvious one is causing a kernel crash (on no!), but there are times you might want trap into the debugger for reasons other than investigating a crash (to set a breakpoint, for example).

add -K to gdb_kdebug#

If you add the -K option to gdb_kdebug in your buildfile then it places a breakpoint on _start in the kernel. This is good if you want to setup a breakpoint really early on.

esh, on#

The embedded shell (esh) has a special prefix that you can use to cause an executable to be launched with the SPAWN_DEBUG flag set. This causes the kernel debugger to set automatically set a breakpoint on the entry point to the user process (see procmgr/procmgr_init.c). That means that once the process has been created and loaded into memory, then the first thing it will do is to trap into the kernel debugger at _start.

On also will do this for you, with the undocumented -D option.

enterkd#

Enterkd is an internal util that does the same thing, but also lets you omit the process name. It does this by executing the special opcode DebugKDBreak(), which will trap into the kernel debugger if present.

Sending a break#

If you are debugging over a serial port, and the appropriate break detect callout has been implemented for the particular device you are using, then you can send a break down the line and at the next timer tick the callout will detect the break condition and trap into the kernel debugger.

Adding DebugKDBreak() to your code#

While not really recommended you can put a DebugKDBreak() into your code to programmatically force a trap into the kernel debugger.

Connecting with gdb#

When the system traps into the kernel debugger, you will see a short message on the debug channel, like

```
KDEBUG at 0xb0334792, (TRAP) S/C/F=5/1/3 in bin/ls
```

This tells you that you have trapped, and gives you the signal, code and fault number, together with the active process and the IP the fault occurred at.

On the kernel debugger port you will see something like

```
||||$S05k#23
```

This is the initiator of the remote protocol, it's now time to connect with gdb.

Connecting with gdb is as simple as pie.

```
# gdb procnto-instr.sym
...
(gdb) set remotebaud 9600
```

(gdb) target remote /dev/ser1

Note that the kernel debugger must already be waiting on the debug port for this to work, and it's important to make sure that you get your serial port settings matching the settings you specified in the build file!

(a common gotcha is setting the baudrate to 9600 in the build file, but then forgetting that devc-ser8250 will default to 57600. If you forgot to disable devc-ser8250 then slay it off but make sure to adjust the baudrate back to 9600 first!)

Also note that if you quit out of gdb whilst connected to the kernel then the system will reboot! This can be avoided by issuing the gdb command _detach_ first.

You can pass the -b 9600 option to gdb to set the baudrate on the command line.

Under windows, the device is call /dev/com1 (for COM1).

A few gotchas to be aware of...#

Debug callouts are often a bit simple#

Debug callouts don't often have a lot of error correction in them. For that reason it's best to choose nice slow baudrates like 9600. You may get away with it at higher rates, but 9600 is normally enough - there's not a huge amount of data to go back and forth.

SMP issues#

Remember that when debugging an SMP kernel the other CPU is still running, even when you're debugging the kernel. It will only stop when it tries to enter the kernel - where it will spin, waiting for you to give it up!

Preventing Preemption Of Kernel Calls#

Of particular note to kernel hackers when connected with gdb, is the -P option which disables preemption of kernel calls.

Imagine that you are set a breakpoint on a kernel call, say ker_connect_attach. You run, someone calls ConnectAttach() and you trap at breakpoint 1. You then think a bit, and then single step. Mysteriously you hit breakpoint 1 again. WTF?!!

This is because of the restartable nature of our kernel calls. Before you have the kernel locked, you can be thrown out at any time (an had better not have changed anything in the global system state!) and restarted if you got preempted by a higher priority thread.

This makes things rather hard when debugging a kernel call though, so for sanities sake (at the expense of realtime determinism), you can and should disable this wonderful feature.

Note that you can also disable it via a global variable in the kernel, named, descriptively, nopreempt.

(gdb) set nopreempt = 1

Debugging Under VMWare or QEMU#

For those of you who are going to debug a kernel that is running on a VMWare Virtual Machine - read THIS, or for QEMU read THIS.