IBM

Inside the Linux kernel debugger

A guide to getting started with KDB

Hariprasad Nellitheertha, Software engineer, IBM

Summary: When debugging kernel problems, being able to trace the kernel execution and examine its memory and data structures is very useful. The built-in kernel debugger in Linux, KDB, provides this facility. In this article you'll learn how to use the features provided by KDB and how to install and set up KDB on a Linux machine. You'll also get acquainted with the commands and the setup and display options available to you in KDB.

Date: 05 Jun 2003
Level: Advanced
Also available in: Japanese

The Linux kernel debugger (KDB) allows you to debug the Linux kernel. This aptly named tool is essentially a patch to the kernel code that lets hackers access kernel memory and data structures. One of the main advantages of KDB is that it does not require an additional machine for debugging: you can debug the kernel that you are running on.

Setting up a machine for KDB requires a little work in that the kernel needs to be patched and recompiled. Users of KDB should be familiar with compiling the Linux kernel (and with kernel internals, to an extent), but refer to the Resources section at the end of this article if you need help compiling the kernel.

In this article we'll start with information on downloading KDB patches, applying them, (re)compiling the kernel, and starting KDB. Then we'll get into the KDB commands and review some of the more often-used commands. Finally, we'll look at some details about the setup and display options.

Getting started

The KDB project is maintained by Silicon Graphics (see Resources for a link), and you will need to download kernel-version dependent patches from its FTP site. The latest version of KDB available (at the time this article was written) is 4.2. You'll need to download and apply two patches. One is the "common" patch that contains the changes for the generic kernel code, and the other is the architecture-specific patch. The patches are available as bz2 files. On an x86 machine running the 2.4.20 kernel, for example, you would need kdb-v4.2-2.4.20-common-1.bz2 and kdb-v4.2-2.4.20-i386-1.bz2.

All examples provided here are for the i386 architecture and the 2.4.20 kernel. You will need to make suitable changes based on your machine and kernel version. You will also need root permissions for performing these operations.

Copy the files into the /usr/src/linux directory and extract the patch files from the bzipped files:

```
#bzip2 -d kdb-v4.2-2.4.20-common-1.bz2

#bzip2 -d kdb-v4.2-2.4.20-i386-1.bz2
```

You will get the kdb-v4.2-2.4.20-common-1 and kdb-v4.2-2.4-i386-1 files.

Now, apply the patches:

```
#patch -p1 <kdb-v4.2-2.4.20-common-1

#patch -p1 <kdb-v4.2-2.4.20-i386-1
```

The patches should apply cleanly. Look for any files ending with .rej. This indicates failed patches. If the kernel tree is clean, the patches should apply without any problems.

Next, the kernel needs to be built with KDB enabled. The first step is to set the `CONFIG_KDB` option. Use your favorite configuration mechanism (xconfig, menuconfig, etc.) to do this. Go to the "Kernel hacking" section at the end and select the "Built-in Kernel Debugger support" option.

There are also a couple of other options you can select based on your preferences. Selecting the "Compile the kernel with frame pointers" option (if present) sets the `CONFIG_FRAME_POINTER` flag. This will lead to better stack tracebacks, as the frame pointer register is used as a frame pointer rather than a general purpose register. You can also select the "KDB off by default" option. This will set the `CONFIG_KDB_OFF` flag and will turn off KDB by default. We'll cover this more in a later section.

Save the configuration and exit. Recompile the kernel. Doing a "make clean" is recommended before building the kernel. Install the kernel in the usual manner and boot to it.

Initializing and setting the environment variables

You can define KDB commands that will be executed during the initialization of KDB. These commands need to be defined in a plain text file called kdb_cmds, which exists in the KDB directory of the Linux source tree (post patching, of course). This file can also be used to define environment variables for setting the display and print options. The comments at the beginning of the file offer help on editing the file. The disadvantage of using this file is that the kernel needs to be rebuilt and reinstalled you change the file.

Activating KDB

If `CONFIG_KDB_OFF` was not selected during compilation, KDB is active by default. Otherwise, you need to activate it explicitly -- either by passing the `kdb=on` flag to the kernel during boot or by doing this once /proc has been mounted:

```
#echo "1" >/proc/sys/kernel/kdb
```

The reverse of the above steps will deactivate KDB. That is, either passing the `kdb=off` flag to the kernel or doing the following will deactivate KDB, if KDB is on by default:

```
#echo "0" >/proc/sys/kernel/kdb
```

There is yet another flag that can be passed to the kernel during boot. The `kdb=early` flag will result in control being passed to KDB very early in the boot process. This would help if you need to debug very early during the boot process.

There are various ways to invoke KDB. If KDB is on, it will get invoked automatically whenever there is a panic in the kernel. Pressing the PAUSE key on the keyboard would manually invoke KDB. Another way to invoke KDB is through the serial console. Of course, to do this, you need to set up the serial console (see Resources for help on this) and you need a program that reads from the serial console. The key sequence Ctrl-A will invoke KDB from the serial console.

KDB commands

KDB is a very powerful tool that allows several operations such as memory and register modification, applying breakpoints, and stack tracing. Based on these, the KDB commands can be classified into several categories. Here are details on the most commonly used commands in each of these categories.

Memory display and modification

The most often-used commands in this category are `md`, `mdr`, `mm`, and `mmW`.

The `md` command takes an address/symbol and a line count and displays memory starting at the address for `line-count` number of lines. If `line-count` is not specified, the defaults as specified by the environment variables are used. If an address is not specified, `md` continues from the last address that was printed. The address is printed at the beginning, and the character conversion is printed at the end.

The `mdr` command takes an address/symbol and a byte count and displays the raw contents of memory starting at the specified address for `byte-count` number of bytes. It is essentially the same as `md`, but it does not display the starting address and the character conversion at the end. The `mdr` command is of very little use.

The `mm` command modifies memory contents. It takes an address/symbol and new contents as parameters and replaces the contents at the address with `new-contents`.

The `mmW` command changes `W` bytes starting at the address. Note that `mm` changes a machine word.

Examples

To display 15 lines of memory starting at 0xc000000:

```
[0]kdb> md 0xc000000 15
```

To change the contents of memory location 0xc000000 to 0x10:

```
[0]kdb> mm 0xc000000 0x10
```

Register display and modification

The commands in this category are `rd`, `rm`, and `ef`.

The `rd` command (without any arguments) displays the contents of the processor registers. It optionally takes three arguments. If the `c` argument is passed, `rd` displays the processor's control registers; with the

`d` argument, it displays the debug registers; and with the `u` argument, the register set of the current task, the last time they entered the kernel, are displayed.

The `rm` command modifies the contents of a register. It takes a register name and `new-contents` as arguments and modifies the register with `new-contents`. The register names depend on the specific architecture. Currently, the control registers cannot be modified.

The `ef` command takes an address as an argument and displays an exception frame at the specified address.

Examples

To display the general register set:

```
[0]kdb> rd
```

To set the contents of register ebx to 0x25:

```
[0]kdb> rm %ebx 0x25
```

Breakpoints

The commonly used breakpoint commands are `bp`, `bc`, `bd`, `be`, and `bl`.

The `bp` command takes an address/symbol as an argument and applies a breakpoint at the address. Execution is stopped and control is given to KDB when this breakpoint is hit. There are a couple of variations of this command that could be useful. The `bpa` command applies the breakpoint on all processors in an SMP system. The `bph` command forces the use of a hardware register on systems that support it. The `bpha` command is similar to the `bpa` command except that it forces the use of a hardware register.

The `bd` command disables a particular breakpoint. It takes in a breakpoint number as an argument. This command does not remove the breakpoint from the breakpoint table but just disables it. The breakpoint numbers start from 0 and are allocated to breakpoints in the order of availability.

The `be` command enables a breakpoint. The argument to this command is also the breakpoint number.

The `bl` command lists the current set of breakpoints. It includes both the enabled and the disabled breakpoints.

The `bc` command removes a breakpoint from the breakpoint table. It takes either a specific breakpoint number as an argument or it takes `*`, in which case it will remove all breakpoints.

Examples

To set up a breakpoint at the function `sys_write()`:

```
[0]kdb> bp sys_write
```

To list all the breakpoints in the breakpoint table:

```
[0]kdb> bl
```

To clear breakpoint number 1:

```
[0]kdb> bc 1
```

Stack tracing

The main stack-tracing commands are `bt`, `btp`, `btc`, and `bta`.

The `bt` command attempts to provide information on the stack for the current thread. It optionally takes a stack frame address as an argument. If no address is provided, it takes the current registers to traceback the stack. Otherwise, it assumes the provided address as a valid stack frame start address and attempts to traceback. If the `CONFIG_FRAME_POINTER` option was set during kernel compilation, the frame pointer register is used to maintain stacks and, hence, the stack traceback can be performed correctly. The `bt` command may not produce correct results in the event of `CONFIG_FRAME_POINTER` not being set.

The `btp` command takes a process ID as an argument and does a stack traceback for that particular process.

The `btc` command does a stack traceback for the running process on each live CPU. Starting from the first live CPU, it does a `bt`, switches to the next live CPU, and so on.

The `bta` command does a traceback for all processes in a particular state. Without any argument, it does a traceback for all processes. Optionally, various arguments can be passed to this command. The processes in a particular state will be processed depending on the argument. The options and the corresponding states are as follows:

- D: Uninterruptible state
- R: Running
- S: Interruptible sleep
- T: Traced or stopped
- Z: Zombie
- U: Unrunnable

Each of these commands prints out a whole lot of information. Check out the Resources below for detailed documentation on these fields.

Examples

To trace the stack for the current active thread:

```
[0]kdb> bt
```

To trace the stack for process with ID 575:

```
[0]kdb> btp 575
```

Other commands

Here are a few other KDB commands that are useful in kernel debugging.

The `id` command takes an address/symbol as an argument and disassembles instructions starting at that address. The environment variable `IDCOUNT` determines how many lines of output are displayed.

The `ss` command single steps an instruction and returns control to KDB. A variation of this instruction is `ssb`, which executes instructions from the current instruction pointer address (printing the instruction on the screen) until it encounters an instruction that would cause a branch. Typical examples of branch instructions are `call`, `return`, and `jump`.

The `go` command lets the system continue normal execution. The execution continues until a breakpoint is hit (if one has been applied).

The `reboot` command reboots the system immediately. It does not bring down the system cleanly, and hence the results are unpredictable.

The `ll` command takes an address, an offset, and another KDB command as arguments. It repeats the command for each element of a linked list. The command executed takes the address of the current element in the list as the argument.

Examples

To disassemble instructions starting from the routine schedule. The number of lines displayed depends on the environment variable `IDCOUNT`:

```
[0]kdb> id schedule
```

To execute instructions until it encounters a branch condition (in this case, instruction `jne`):

```
[0]kdb> ssb

0xc0105355 default_idle+0x25: cli
0xc0105356 default_idle+0x26: mov 0x14(%edx),%eax
0xc0105359 default_idle+0x29: test %eax, %eax
0xc010535b default_idle+0x2b: jne 0xc0105361 default_idle+0x31
```

Tips and tricks

Debugging a problem involves locating the source of the problem using a debugger (or any other tool) and using the source code to track the root cause of the problem. Using source code alone to determine problems is extremely difficult and may be possible only for expert kernel hackers. On the other hand, newbies tend to rely excessively on debuggers to fix bugs. This approach may lead to incorrect solutions to problems. The fear is that such an approach leads to fixing symptoms rather than the actual problems. A classic example of such a mistake is adding error handling code to take care of NULL pointers or bad references without looking into the real cause of the invalid references.

The dual approach of studying the code and using debugging tools is the best way to identify and fix problems.

The primary use of debuggers is to get to the location of the bug, confirm the symptom (and cause, in some cases), determine the values of variables, and determine how the program got there (that is, establish the call stack). An experienced hacker will know which debugger to use for a particular kind of problem and will quickly obtain the necessary information from debugging and get on with analyzing the code to identify the cause.

Here, then, are some tips for you to achieve the above mentioned results quickly using KDB. Of course, keep in mind that speed and accuracy in debugging comes with experience, practice, and good knowledge of the system (hardware, kernel internals, etc.)

Tip #1

In KDB, typing an address at the prompt returns its nearest symbol match. This is extremely useful in

stack analysis and in determining the addresses/values of global data and function addresses. Similarly, typing the name of a symbol returns its virtual address.

Examples

To indicate that the function `sys_read` starts at address 0xc013db4c:

```
[0]kdb> 0xc013db4c

0xc013db4c = 0xc013db4c (sys_read)
```

Similarly,

Similarly, to indicate that `sys_write` is at address 0xc013dcc8:

```
[0]kdb> sys_write

sys_write = 0xc013dcc8 (sys_write)
```

These help in locating global data and function addresses while analyzing stacks.

Tip #2

Whenever present, use the `CONFIG_FRAME_POINTER` option while compiling the kernel with KDB. To do this, you need to select the "Compile the kernel with frame pointers" option under the "Kernel hacking" section while configuring the kernel. This ensures that the frame pointer register will be used as a frame pointer leading to accurate tracebacks. You could, in fact, manually dump the contents of the frame pointer register and trace the entire stack. For example, on an i386 machine, the %ebp register can be used to traceback the entire stack.

For example, after executing the first instruction at function `rmqueue()`, the stack looked as follows:

```
[0]kdb> md %ebp

0xc74c9f38 c74c9f60 c0136c40 000001f0 00000000
0xc74c9f48 08053328 c0425238 c04253a8 00000000
0xc74c9f58 000001f0 00000246 c74c9f6c c0136a25
0xc74c9f68 c74c8000 c74c9f74 c0136d6d c74c9fbc
0xc74c9f78 c014fe45 c74c8000 00000000 08053328

[0]kdb> 0xc0136c40

0xc0136c40 = 0xc0136c40 (__alloc_pages +0x44)

[0]kdb> 0xc0136a25

0xc0136a25 = 0xc0136a25 (_alloc_pages +0x19)

[0]kdb> 0xc0136d6d

0xc0136d6d = 0xc0136d6d (__get_free_pages +0xd)
```

We can see that `rmqueue()` has been called by `__alloc_pages`, which in turn has been called by `_alloc_pages`, and so on.

The first double word of every frame points to the next frame and this is immediately followed by the address of the calling function. Hence, tracing the stack becomes an easy job.

Tip #3

The `go` command, optionally, takes an address as a parameter. If you want to continue execution at a particular address, you could provide that address as a parameter. The other alternative is to modify the instruction pointer register using the `rm` command and just type `go`. This would be useful if you wanted to skip a particular instruction or a set of instructions that appeared to be causing a problem. Note, though, that using this instruction without care could lead to severe problems, and the system might crash badly.

Tip #4

You can define your own set of commands with a useful command called `defcmd`. For example, whenever you hit a breakpoint, you might wish to simultaneously check a particular variable, check the contents of some registers, and dump the stack. Normally, you would have to type a series of commands to be able to do this all at the same time. The `defcmd` allows you to define your own command, which could consist of one or more of the predefined KDB commands. Only one command would then be needed to do all three jobs. The syntax for this is as follows:

```
[0]kdb> defcmd name "usage" "help"

[0]kdb> [defcmd] type the commands here

[0]kdb> [defcmd] endefcmd
```

For example, a (trivial) new command called `hari` could be defined that would display one line of memory starting at address 0xc000000, display the contents of the registers, and dump the stack:

```
[0]kdb> defcmd hari "" "no arguments needed"

[0]kdb> [defcmd] md 0xc000000 1

[0]kdb> [defcmd] rd

[0]kdb> [defcmd] md %ebp 1

[0]kdb> [defcmd] endefcmd
```

The output of this command would be:

```
[0]kdb> hari

[hari]kdb> md 0xc000000 1

0xc000000 00000001 f000e816 f000e2c3 f000e816

[hari]kdb> rd

eax = 0x00000000 ebx = 0xc0105330 ecx = 0xc0466000 edx = 0xc0466000
```

```
....
...

[hari]kdb> md %ebp 1

0xc0467fbc c0467fd0 c01053d2 00000002 000a0200

[0]kdb>
```

Tip #5

The `bph` and `bpha` commands can be used (provided the architecture supports use of hardware registers) to apply read and write breakpoints. This means we can get control whenever data is read from or written into a particular address. This can be extremely handy when debugging data/memory corruption problems, where you can use it to identify the corrupting code/process.

Examples

To enter the kernel debugger whenever four bytes are written into address 0xc0204060:

```
[0]kdb> bph 0xc0204060 dataw 4
```

To enter the kernel debugger when at least two bytes of data starting at 0xc000000 are read:

```
[0]kdb> bph 0xc000000 datar 2
```

Conclusion

KDB is a handy and powerful tool for performing kernel debugging. It offers various options and enables analysis of memory contents and data structures. Best of all, it does not require an additional machine to perform the debugging.

Resources

- Find the KDB man pages in the Documentation/kdb directory.
- For information on setting up the serial console, look for serial-console.txt in the Documentation directory.
- Download KDB at SGI's kernel debugger project Web site.
- For an overview of several scenario-based debugging techniques for Linux, read "Mastering Linux debugging techniques" (*developerWorks*, August 2002).
- Those looking for information on debugging OS/2 should read Volume II of the four-volume IBM Redbook, *The OS/2 Debugging Handbook*.
- Find more resources for Linux developers in the *developerWorks* Linux zone.

About the author

Hariprasad Nellitheertha works in IBM's Linux Technology Center in Bangalore, India. He is currently working on the Linux Change Team fixing kernel and other Linux bugs. Hari has worked on the OS/2 kernel and file systems. His interests include Linux kernel internals, file systems, and autonomic computing. You can contact Hari at nharipra@in.ibm.com.