KGDB Documentation for versions 2.3 & 2.4

Table of Contents

Introduction

KGDB is a source level debugger for linux kernel. It is used along with gdb to debug linux kernel. Kernel developers can debug a kernel similar to application programs with the use of KGDB. It makes it possible to place breakpoints in kernel code, step through the code and observe variables.

Two machines are required for using KGDB. One of these machines is a development machine and the other is a test machine. The machines are connected through a serial line, a null-modem cable which connects their serial ports. Recent versions of KGDB work over ethernet also.The kernel to be debugged runs on the test machine. gdb runs on the development machine. The serial line is used by gdb to communicate to the kernel being debugged.

KGDB is available for i386, x86_64, ppc, arm, mips and ia64 architectures. KGDB 2.3 and 2.4 (2.6.13 and 2.6.15.5 respectively) also work fine over ethernet interface.

KGDB is a kernel patch. It has to be applied to a linux kernel to enable kernel debugging. KGDB patch adds following components to a kernel

- **gdb stub -** The gdb stub is heart of the debugger. It is the part that handles requests coming from gdb on the development machine. It has control of all processors in the target machine when kernel running on it is inside the debugger.
- **modifications to fault handlers** - Kernel gives control to debugger when an unexpected fault occurs. A kernel which does not contain gdb panics on unexpected faults. Modifications to fault handlers allow kernel developers to analyze unexpected faults.
- **serial communication** - This component uses a serial driver in the kernel and offers an interface to gdb stub in the kernel. It is responsible for sending and receiving data from a serial line. This component is also responsible for handling control break request sent by gdb.

KGDB is available for x86 architecture on several versions of linux kernel from 2.4.6 to 2.6.15. Please go to downloads page for getting it.

Using KGDB

- ## Requirements
  - ### Hardware Requirements:

    Two x86 machines are required for using KGDB. One of the machines runs a kernel to be debugged. The other machine runs gdb. The machine that runs the kernel to be debugged is called the Test machine while the machine that runs gdb is called the Development machine.

    A single development machine can be used with several test machines. The architecture of development and test machine can be different. But the architecture of test machine and the architecture of the kernel being debugged should match. The development machine will run a copy of GDB per test machine. The development machine should have at least 128MB RAM so that loading debugging info into gdb does not result into too much of swap space usage.

    A serial line is required between the development and the test machine. For the serial line to be established, machines need one serial port each. A null modem cable is required to connect serial ports of the machines. Recent versions of KGDB work over ethernet also. If KGDB is run over ethernet, a serial line is not required. If debugging of modules is needed, the two machines should be connected through a network. The network connection is required by rcp and rsh commands used by module debugging utilities. KGDB itself does not need a network connection. It needs a serial line only. It's also convenient to have the machines connected through a network for looking into the test machine. The documentation on this site assumes presence of a network.

  - ### Software Requirements:

    Both development and test machines require redhat 7.3 or later. The test machine runs a kernel to be debugged. For example to debug a linux kernel version 2.6.13, the test machine will run 2.6.13 kernel. The development machine need not run any special kernel (the kernel that comes with rh9 will do).

  - ### Hardware Setup:

    The setup required for running KGDB over a serial line is described below. KGDB over ethernet doesn't require this setup.

    Null-modem cable:

    Connect the machines using a null-modem cable. A null-modem cable is a 3 wire cable connecting serial ports of the machines. It has DB9 or DB25 connectors at the end to be plugged into serial ports. Connections for DB25 connectors at both

the ends are shown below.
Connector 1 pins - Connector 2 pins
2 (TxD) - 3 (RxD)
3 (RxD) - 2 (TxD)
7 (GND) - 7 (GND)

Serial line transmission rate:

Serial ports support transmission rates from 110 baud to 115200 baud. Baud rates supported by a serial port depend on the chipset used for the serial port. Default baud rate for a serial port is 9600. Higher baud rates result in faster communication between a test kernel and gdb, hence are preferred over lower baud rates.

A serial port may support supports all these rates but the null-modem cable may not be able to support them. It is recommended that you check the maximum speed supported by the serial port and the null-modem cable and use it for kgdb. For doing analysis of threads from gdb, a baud rate of 115200 is recommended as lower rates require a long time to get a list of threads from a test kernel.

• Preparing a kernel

○ Preparing a kernel:

To prepare a kernel for testing, apply a KGDB patch to a linux kernel source. Then enable Remote (serial) kernel debugging with GDB from kernel hacking, which will enable KGDB code in the kernel. This will enable choice of a few more config options with change KGDB behavior.

These are described below:-

Thread analysis: With thread analysis enabled, gdb can talk to kgdb stub to list threads and to get stack trace for a thread. This option also enables some code which helps gdb get exact status of thread. Thread analysis adds some overhead to *schedule* and *down* functions. You can disable this option if you do not want to compromise on execution speed.

Console messages through GDB: With this option enabled, kgdb stub will route console messages through GDB. Console messages from the test machine will appear in a terminal on the development machine where gdb is running. Other consoles will not be affected by this option. After the kernel is configured, build it and add it to GRUB. KGDB stub requires following options in a kernel command line.

kgdbwait: This option causes KGDB to wait for a GDB connection during kernel bootup.

kgdb8250=<port number>,<port speed>

Where port number can be 0 to 3 for ports ttyS0(COM1) to ttyS3(COM4) respectively. Supported port speeds are 9600, 19200, 38400, 57600 and 115200.

An example of above procedure is shown below
On development machine:
1. Extract a kernel source
   $ cd ${BASE_DIR}
   $ tar -jxvf linux-2.6.15.5.tar.bz2
2. Unzip the kgdb patch
   $ tar -jxvf linux-2.6.15.5-kgdb-2.4.tar.bz2
3. Change the directory as follows:
   $ cd ${BASE_DIR}/linux-2.6.15.5
4. Apply KGDB patches
   $ patch -p1 < ${BASE_DIR}/linux-2.6.15.5-kgdb-2.4/core-lite.patch
   .....
   $ patch -p1 < ${BASE_DIR}/linux-2.6.15.5-kgdb-2.4/i386.patch
   Follow the order mentioned in "series" file while applying the patches
5.Configure the kernel
   $ make xconfig or make menuconfig
Configure drivers and other kernel options
To enable kgdb, enable following config options (in this order).
   Kernel hacking ->
   [*] KGDB: kernel debugging with remote gdb ->
   [*] KGDB: Console messages through gdb
   Method for KGDB communication (KGDB: On generic serial port (8250)) --->
      ( ) KGDB: Use only kernel modules for I/O
      (X) KGDB: On generic serial port (8250)
      ( ) KGDB: On ethernet - in kernel
   --- KGDB: On generic serial port (8250)
   [*] Simple selection of KGDB serial port
      (115200) Debug serial port baud rate
      (0) Serial port number for KGDB
6.Build the kernel
   $ make bzImage
7.Copy the kernel to target machine
It is assumed that the user who is working on the development machine has rsh permissions on the test machine for root account.
   $ rcp System.map testmach:/boot/System.map-2.6.15.5-kgdb
   $ rcp arch/i386/boot/bzImage testmach:/boot/vmlinuz- 2.6.15.5-kgdb

On test machine:
8.Add a grub entry:
   title 2.6.15.5-kgdb

root (hd0,0)

kernel /boot/vmlinuz-2.6.15.5-kgdb ro root=/dev/hda1 kgdbwait kgdb8250=1,115200

Modules in the test machine:

It is recommended that you compile all drivers in the kernel itself instead of compiling them as a module.

Connecting to debug kernel:

Set the speed of the serial line on development machine to the value you have given to KGDB kernel on the test machine. Start GDB from the kernel source directory giving it the vmlinux file on command line as the object file. Run the KGDB kernel on the test machine and wait till it prints a message:

Uncompressing Linux... Ok, booting the kernel.

Then connect to the target machine from GDB using *target remote* command. This command needs the serial line path to be specified as an argument. At this point GDB and the KGDB stub will be connected and GDB will have a control of the target kernel. You can now use GDB commands for inserting breakpoints, printing values of variables, breaking and continuing execution.

An example of above procedure is given below:

On development machine:

1.Set appropriate speed for serial line.

$ stty ispeed 115200 ospeed 115200 < /dev/ttyS0

2.Start GDB. It will take some time because vmlinux contains a lot of debugging information. If you want to debug modules also, you'll need to download the GDB from this site. Please refer to module debugging setup for details. I have placed it at the path /usr/local/bin/gdbmod-2.3 on my machine. I change the name so that I don't use that gdb accidentally for application debugging also. You can use the default gdb that comes with redhat installation provided you don't want module debugging.

cd ${BASE_DIR}/linux-2.6.15.5

<root#> gdb ./vmlinux

GNU gdb (GDB) 7.1-ubuntu

Copyright (C) 2010 Free Software Foundation, Inc.

License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software: you are free to change and redistribute it.

There is NO WARRANTY, to the extent permitted by law. Type "show copying" and "show warranty" for details.

This GDB was configured as "i486-linux-gnu".

For bug reporting instructions, please see:

<http://www.gnu.org/software/gdb/bugs/>...
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
breakpoint () at kernel/kgdb.c:1212
1212 atomic_set(&kgdb_setting_breakpoint, 0);
(gdb)

On test machine:
3.Select kgdb kernel from GRUB prompt The kernel will start and after doing some initializations, wait for gdb to connect. It will write following prompt on the console:
Uncompressing Linux... Ok, booting the kernel.

On development machine:
4.Connect to the test machine using gdb command "target remote"
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
breakpoint () at gdbstub.c:1153
1153 }
(gdb)
5.Now gdb is connected to the kernel. It is waiting for a command. Use the continue command to let the test kernel proceed.
(gdb) c
Continuing.
The test kernel continues here and the system boots as usual. If console messages through gdb was selected while configuring the kernel, console log will appear here from gdb as follows:
PCI: PCI BIOS revision 2.10 entry at 0xfb230, last bus=0
PCI: Using configuration type 1
PCI: Probing PCI hardware
Limiting direct PCI/PCI transfers.
Activating ISA DMA hang workarounds.
isapnp: Scanning for PnP cards...
isapnp: No Plug & Play device found
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Starting kswapd v1.8
Now gdb is connected to the test kernel. If a kernel panic situation occurs, instead

of declaring a panic, the kernel will first give control to gdb so that the situation can be analyzed.

Preparing modules for debugging

Requirements and preparation on development and test machines

(Module debugging setup)

GDB on development machine

Install on the development machine, a GDB containing module debugging features. It's sources are available from the kgdb downloads page.This gdb is derived from GDB version 6.0. It contains all features of GDB 6.0 plus ability to automatically detect module loading and unloading. You can either download the sources, build them and install the GDB at /usr/local/bin/gdbmod-2.3. Prebuilt version of GDB are also available from the downloads page.

No special setup is needed on test machines. Modules can be present in a test machine root filesystem or a ramdisk.

Loading the module symbols in GDB

This step is to be done on development machine. It loads the symbols of the module in GDB and makes them available to GDB for debugging. One must use the special GDB for debugging.

Its available on the this link (http://kgdb.geeksofpune.in/downloads/gdbmod-2.3.bz2).

NOTE: The module has to be built by debugging information.

#cd /usr/src/linux- 2.6.15.5
#gdbmod-2.3 vmlinux
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
breakpoint () at gdbstub.c:1153
1153 }
(gdb)set solib-search-path /usr/linux-2.6.15.5/drivers/net

gdb has to be able to locate module files once kgdb informs it that a module has been loaded. For this you tell it the path to those files using a command "set solib-search-path"

e.g.

"set solib-search-path /home/mayur/work/mymodule", where a module file is located at /home/mayur/work/mymodule/mymodule.ko

or

"set solib-search-path /home/mayur/work/mymodule:/usr/src/linux-2.6.15.5/drivers/net"

Inserting the module in the kernel

Use 'insmod' command for inserting the module on testmachine.
GDB searches for a module file after appending a ".o" or ".ko" to the module name as we see in "lsmod" output.

 # insmod mymodule.ko

Now module symbols are loaded and they can be debugged as normal kernel code.

Debugging Kernel

Using GDB for kernel debugging

Usage of GDB for kernel debugging is similar to that for debugging application processes. These pages briefly describe gdb commands and kernel issues related to these commands. Most of the kernel issues are related to inline functions used in the kernel. KGDB supports GDB execution control commands, stack trace and thread analysis. It doesn't support GDB watchpoints. kgdb watchpoints are accessed through GDB macros. Use of these macros is also described in these webpages. Please refer to GDB documenation for a list of GDB commands and explanations of their usage. GDB documenation can be accessed by the command:

info gdb

GDB also has online help. The GDB command help gives a short description of a command name or topic name given as an argument.

Killing of terminating GDB

If GDB does not respond to user input, you can kill it. If the test kernel was controlled by kgdb when gdb was killed, a new GDB can be started and it can be connected to kgdb. Otherwise you'll first need to make the test kernel drop into kgdb. To do this send an ascii 3 character into the serial line. An example of a command which does that is:

echo -e "\003" > /dev/ttyS0

When GDB has printed a command prompt, you can quit it and start a new GDB and connect to KGDB immediately.
GDB removes all breakpoints before presenting a command prompt. Hence if a new GDB is started, there will be no breakpoints in the kernel. You'll need to add all the breakpoints again. However, if a running GDB was killed after adding some breakpoints, they will remain in the kernel. New GDB will not know about those breakpoints. Hence if one of these breakpoints is hit during subsequent execution, GDB will not be able to handle it properly. Since a breakpoint modifies code, execution after this

point is unpredictable. A hardware reset is recommended in this case. KGDB versions 1.6 onwards maintain breakpoints within the stub itself, hence this problem is not seen.
KGDB hardware watchpoints are handled by KGDB stub and GDB does not know anything about them. Hence there are no issues with watchpoints if GDB is killed.

- **Test Machine Reboot**
  If a machine is rebooted, it is recommended that you terminate or kill GDB before the debug kernel starts again. If that is not done and kgdb connects to GDB and if some breakpoints were added in the previous execution of the kernel, GDB will try to remove them on connection. Since the breakpoints will be absent in the rebooted kernel, the result is undesirable.

Controlling the kernel execution

- **Stopping the kernel execution**
  To stop kernel execution, press *Ctrl + C* on the gdb terminal. GDB will send a stop message to KGDB stub, which will take control of the kernel and contact GDB. GDB will then present a command prompt and wait for user input. At this point all processors will be controlled by KGDB and no other part of the kernel will be executing. You can now use any GDB commands. GDB will present and prompt and wait for a command till you tell it to continue execution.
- **Continuing kernel execution**
  To continue kernel execution, use GDB command continue. gdb will tell the KGDB stub to continue kernel execution. Kernel execution will continue after this point. GDB will not expect a user command till the user either breaks execution using *Ctrl + C*, or KGDB stub gives control to it because of a breakpoint or some other reason.
- **Breakpoints**
  Use gdb *break* command to stop kernel execution at a function or a source code line. The command accepts a function name or a source code file name appended with a : and a line number as an argument.
- **Stepping through the code**
  To step into a code statement, use gdb command *step*. This command will run the kernel for one statement and again hand over the control to GDB. This command will step into function calls also. If you want to skip stepping into function calls, use gdb command *next*. With both the commands kgdb continues kernel execution on all the processors. With the

step command, gdb causes kgdb to do a one instruction at a time stepping till next code statement is reached. kgdb has to catch all processors and release them on every instruction step. If the code statement is in a loop, *step* may take a long time. For example stepping into *copy_to_user* function for copying a large buffer will usually require a few minutes. The same issue is also present with the *next* command as well.

Stack Trace

Once GDB is in command mode, you can look at the stack trace using backtrace command. This command shows a list of function calls starting from the function where a system call entered the kernel. For each function, it prints the source code file name and line number where next function was called and for the innermost function, where execution stopped. It also prints argument values to these functions. We can break the debugger by pressing *Ctrl + C,* and see a stack trace as follows:

(gdb) backtrace
#0 breakpoint () at gdbstub.c:1160
#1 0xc0188b6c in gdb_interrupt (irq=3, dev_id=0x0, regs=0xc02c9f9c)
at gdbserial.c:143
#2 0xc0108809 in handle_IRQ_event (irq=3, regs=0xc02c9f9c,
action=0xc12fd200)
at irq.c:451
#3 0xc0108a0d in do_IRQ (regs={ebx = -1072672288, ecx = 0, edx = -
1070825472,
esi = -1070825472, edi = -1072672288, ebp = -1070817328, eax = 0,
xds = -1072693224, xes = -1072693224, orig_eax = -253,
eip = -1072672241, xcs = 16, eflags = 582, esp = -1070817308,
xss = -1072672126}) at irq.c:621
#4 0xc0106e04 in ret_from_intr () at af_packet.c:1878
#5 0xc0105282 in cpu_idle () at process.c:135
#6 0xc02ca91f in start_kernel () at init/main.c:599
#7 0xc01001cf in L6 () at af_packet.c:1878
Cannot access memory at address 0x8e000

Unless number of stack frames to be printed is specified as an argument to the backtrace command, gdb stops printing backtrace only when the stack trace goes out of accessible address space. The function call hierarchy as shown above is *ret_from_intr, do_IRQ, handle_IRQ_event, gdb_interrupt*.
Let's place a breakpoint in *ext2_readlink* and access a symbolic link so that the

breakpoint is hit.

(gdb) br ext2_readlink
Breakpoint 2 at 0xc0158a05: file symlink.c, line 25.
(gdb) c
Continuing.

On running command *ls -l /boot/vmlinuz* on the test machine,

Breakpoint 2, ext2_readlink (dentry=0xc763c6c0, buffer=0xbfffed84 "\214\005", buflen=4096) at symlink.c:25
*25 char *s = (char *)dentry->d_inode->u.ext2_i.i_data;*
(gdb) bt
#0 ext2_readlink (dentry=0xc763c6c0, buffer=0xbfffed84 "\214\005", buflen=4096) at symlink.c:25
#1 0xc013b027 in sys_readlink (path=0xbfffff77 "/boot/vmlinuz",
buf=0xbfffed84 "\214\005", bufsiz=4096) at stat.c:262
#2 0xc0106d83 in system_call () at af_packet.c:1878
#3 0x804aec8 in ?? () at af_packet.c:1878
#4 0x8049697 in ?? () at af_packet.c:1878
#5 0x400349cb in ?? () at af_packet.c:1878

GDB has printed some invalid stackframes in above backtrace. This is because GDB doesn't know where to stop a backtrace. We can ignore the stackframes 3 to 5 as they are invalid. The system call *readlink* entered the kernel at *system_call* function. The function is shown in *af_packet.c* which is incorrect. GDB is not able to figure out the correct code line, because it's a function in an assembly language file. GDB can handle inline assembly in C correctly. Further hierarchy is *sys_readlink* and *ext2_readlink.*

After this we remove the breakpoint with delete command and continue.

(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) c
Continuing.

○ Inline Functions

Information printed in a GDB backtrace is usually sufficient to find out what a function call hierarchy is at the point where the execution stopped. It may not be enough when one of the stack frames is inside an expanded inline function. Since inline functions are expanded inline, if execution stopped inside an inline function, or if a call to an inner function was made from an inline function, GDB shows source code file name and line number of the statement in the inline

function. It may be possible to know which function the inline function was called from and where it was called by looking at the outer function. If the inline function was called two times, or if it is not possible to know which function the inline function was called from, following procedure can be used to find out this information.

gdb also shows code addresses alongwith function names in a backtrace. The statement where an inline function was called can be found out from these code addresses. The disasfun.sh script can be used here to disassemble a kernel function from the *vmlinux* file with source code references. The vmlinux file contains absolute addresses of kernel functions, hence the addresses seen in the assembly code are the addresses in memory. An example of how this can be done is given below.

KGDB thread analysis *(CONFIG_KGDB_THREAD)* is enabled while configuring the kernel. GDB is connected to the target kernel.

Let's break it using *Ctrl+C*, place a breakpoint in function *__break* and continue.

Program received signal SIGTRAP, Trace/breakpoint trap.

breakpoint () at gdbstub.c:1160

1160 }

(gdb) break __down

Breakpoint 1 at 0xc0105a43: file semaphore.c, line 62.

(gdb) c

Continuing.

To hit the breakpoint, run man lilo on the target machine. The breakpoint will be hit and gdb will go in command mode.

Breakpoint 1, __down (sem=0xc7393f90) at semaphore.c:62

62 add_wait_queue_exclusive(&sem- >wait, &wait);

(gdb) backtrace

#0 __down (sem=0xc7393f90) at semaphore.c:62

#1 0xc0105c70 in __down_failed () at af_packet.c:1878

#2 0xc011433b in do_fork (clone_flags=16657, stack_start=3221199556, regs=0xc7393fc4, stack_size=0)

at /mnt/work/build/old- pc/linux-2.4.6-kgdb/include/asm/semaphore.h:120

#3 0xc010594b in sys_vfork (regs={ebx = 1074823660, ecx = 1074180970, edx = 1074823660, esi = -1073767732, edi = 134744856, ebp = -1073767712, eax = 190, xds = 43, xes = 43, orig_eax = 190, eip = 1074437320, xcs = 35, eflags = 518, esp = -1073767740, xss = 43}) at process.c:719

The line number in function sys_vfork is correctly shown 719 in file process.c We can confirm that by listing the code around this line number.

(gdb) list process.c:719

*714 * do not have enough call-clobbered registers to hold all*
*715 * the information you need.*
*716 */*
717 asmlinkage int sys_vfork(struct pt_regs regs)
718 {
719 return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs.esp, ®s, 0);
720 }
721
*722 /**
*723 * sys_execve() executes a new program.*

As shown by gdb, *do_fork* function is called from *sys_vfork* function. Let's consider frame 2 in the stack trace. gdb shows that it's on line number 120 in file *semaphore.h* This is correct, though not very useful.

(gdb) list semaphore.h:118
113 */
114 static inline void down(struct semaphore * sem)
115 {
116 #if WAITQUEUE_DEBUG
117 CHECK_MAGIC(sem- >__magic);
118 #endif
119
120 __asm__ __volatile__(<-----
121 "# atomic down operation\n\t"
122 LOCK "decl %0\n\t" /* --sem->count */

The only information we get is that it's inside an inline expanded *down* function in *do_fork* at the statement indicated by an arrow. gdb has also printed the absolute address of the code in *do_fork* from next function was called: *0xc011433b*. Here we use the *disasfun* script to find which line of code this address corresponds to. Part of the output of the command disasfun *vmlinux do_fork* is shown below:

if ((clone_flags & CLONE_VFORK) && (retval > 0))
c011431d: 8b 7d 08 mov 0x8(%ebp),%edi
c0114320: f7 c7 00 40 00 00 test $0x4000,%edi
c0114326: 74 13 je c011433b <do_fork+0x707>
c0114328: 83 7d d4 00 cmpl $0x0,0xffffffd4(%ebp)
c011432c: 7e 0d jle c011433b <do_fork+0x707>
#if WAITQUEUE_DEBUG
CHECK_MAGIC(sem->__magic);
#endif
__asm__ __volatile__(

c011432e: 8b 4d d0 mov 0xffffffd0(%ebp),%ecx
c0114331: f0 ff 4d ec lock decl 0xffffffec(%ebp)
c0114335: 0f 88 68 95 13 00 js c024d8a3 <stext_lock+0x7bf> down(&sem);
return retval;
c011433b: 8b 45 d4 mov 0xffffffd4(%ebp),%eax <----
c011433e: e9 8d 00 00 00 jmp c01143d0 <do_fork+0x79c>
Looking at the code in fork.c we know where above code is:
fork_out:
if ((clone_flags & CLONE_VFORK) && (retval > 0))
down(&sem)

- ## Thread Analysis

GDB features include analysis of application threads. GDB provides a listing of threads created by an application program. It then allows a developer to look into any of those threads. This GDB feature can be used with KGDB to look into kernel threads. GDB can provide a listing of all the threads in a kernel. A developer can specify a particular thread to be analyzed. GDB commands like *backtrace, info regi* then show the information in context of the specified thread. All threads created by an application share the same address space. Similarly all kernel threads share kernel address space. User address space for each kernel thread may be different. Hence gdb thread applies well to analysis of kernel code and data structures that reside in kernel address space.
gdb info pages give more information on using GDB thread analysis feature. An example of kernel thread analysis is shown below:
The gdb command info threads gives a listing of kernel threads.
(gdb) info thr
21 thread 516 schedule_timeout (timeout=2147483647) at sched.c:411
20 thread 515 schedule_timeout (timeout=2147483647) at sched.c:411
19 thread 514 schedule_timeout (timeout=2147483647) at sched.c:411
18 thread 513 schedule_timeout (timeout=2147483647) at sched.c:411
17 thread 512 schedule_timeout (timeout=2147483647) at sched.c:411
16 thread 511 schedule_timeout (timeout=2147483647) at sched.c:411
15 thread 438 schedule_timeout (timeout=2147483647) at sched.c:411
14 thread 420 schedule_timeout (timeout=- 1013981316) at sched.c:439
13 thread 406 schedule_timeout (timeout=-1013629060) at sched.c:439
12 thread 392 do_syslog (type=2, buf=0x804dc20 "run/utmp", len=4095)
at printk.c:182
11 thread 383 schedule_timeout (timeout=2147483647) at sched.c:411
10 thread 328 schedule_timeout (timeout=2147483647) at sched.c:411
9 thread 270 schedule_timeout (timeout=- 1011908724) at sched.c:439

8 thread 8 interruptible_sleep_on (q=0xc02c8848) at sched.c:814
7 thread 6 schedule_timeout (timeout=- 1055490112) at sched.c:439
6 thread 5 interruptible_sleep_on (q=0xc02b74b4) at sched.c:814
5 thread 4 kswapd (unused=0x0) at vmscan.c:736
4 thread 3 ksoftirqd (__bind_cpu=0x0) at softirq.c:387
3 thread 2 context_thread (startup=0xc02e93c8) at context.c:101
2 thread 1 schedule_timeout (timeout=- 1055703292) at sched.c:439
** 1 thread 0 breakpoint () at gdbstub.c:1159*
(gdb)

gdb assigns its own id to each thread as shown above. When referring to a thread inside gdb, this id is used. For example thread 7 (PID 7) has gdb id 8. To analyze the kernel thread 8, we specify thread 9 to gdb. gdb then switches to this thread for further analysis. Further commands like *backtrace* apply to this thread.

(gdb) thr 9
[Switching to thread 9 (thread 270)]
#0 schedule_timeout (timeout=-1011908724) at sched.c:439
439 del_timer_sync(&timer);
(gdb) bt
#0 schedule_timeout (timeout=-1011908724) at sched.c:439
#1 0xc0113f36 in interruptible_sleep_on_timeout (q=0xc11601f0, timeout=134)
at sched.c:824
#2 0xc019e77c in rtl8139_thread (data=0xc1160000) at 8139too.c:1559
#3 0xc010564b in kernel_thread (fn=0x70617773, arg=0x6361635f, flags=1767859560) at process.c:491
#4 0x19 in uhci_hcd_cleanup () at uhci.c:3052
#5 0x313330 in ?? () at af_packet.c:1891
Cannot access memory at address 0x31494350
(gdb) info regi
eax 0xc38fdf7c -1013981316
ecx 0x86 134
edx 0xc0339f9c -1070358628
ebx 0x40f13 266003
esp 0xc3af7f74 0xc3af7f74
ebp 0xc3af7fa0 0xc3af7fa0
esi 0xc3af7f8c -1011908724
edi 0xc3af7fbc -1011908676
eip 0xc011346d 0xc011346d
eflags 0x86 134
cs 0x10 16

ss 0x18 24
ds 0x18 24
es 0x18 24
fs 0xffff 65535
gs 0xffff 65535
fctrl 0x0 0
fstat 0x0 0
ftag 0x0 0
fiseg 0x0 0
fioff 0x0 0
foseg 0x0 0
fooff 0x0 0
---Type <return> to continue, or q <return> to quit---
fop 0x0 0
(gdb) thr 7
[Switching to thread 7 (thread 6)]
#0 schedule_timeout (timeout=-1055490112) at sched.c:439
439 del_timer_sync(&timer);
(gdb) bt
#0 schedule_timeout (timeout=-1055490112) at sched.c:439
#1 0xc0137ef2 in kupdate (startup=0xc02e9408) at buffer.c:2826
#2 0xc010564b in kernel_thread (fn=0xc3843a64, arg=0xc3843a68, flags=3280222828) at process.c:491
#3 0xc3843a60 in ?? ()
Cannot access memory at address 0x1f4
(gdb)

Process information macros ps and psname available from the downloads page are useful for thread analysis.The ps macro provides a names and ids of threads running in a kernel.

(gdb) ps
0 swapper
1 init
2 keventd
3 ksoftirqd_CPU0
4 kswapd
5 bdflush
6 kupdated
8 khubd
270 eth0
328 portmap

383 syslogd
392 klogd
406 atd
420 crond
438 inetd
511 mingetty
512 mingetty
513 mingetty
514 mingetty
515 mingetty
516 mingetty
(gdb)

The psname macro can be used to get name of a thread when it's id is known.

(gdb) psname 8
8 khubd
(gdb) psname 7
(gdb)

- ## Caveats in using GDB thread model for kernel threads
  - GDB may require a really large amount of time to show thread listing on a slow serial line. It can go even over few tens of seconds if the number of threads is over hundred.
  - GDB assumes that thread ids always increase. GDB queries KGDB for threads that have been created with ids greater than the largest thread id in previous thread listing. Because thread ids wrap around, this condition may be violated in case of a heavily loaded system which has been running for a long time. If you think that the test kernel you are using may have wrapped around thread ids, it's necessary to quit from GDB and restart it when you want to get a thread listing. This gets around the problem of GDB remembering the largest thread id from previous thread listing.
- ## Watchpoints

  KGDB stub contains support for hardware breakpoints using debugging features of ia-32(x86) processors. These breakpoints do not need code modification. They use debugging registers. 4 hardware breakpoints are available in ia-32 processors. Each hardware breakpoint can be of one of the following three types.
  - **Execution Breakpoint** - An Execution breakpoint is triggered when code at the breakpoint address is executed. As limited number of hardware breakpoints are available, it is advisable to use software breakpoints (break command) instead of execution

hardware breakpoints, unless modification of code is to be avoided.

- **Write Breakpoint** - A write breakpoint is triggered when memory location at the breakpoint address is written. A write breakpoint can be placed for data of variable length. Length of a write breakpoint indicates length of the datatype to be watched. Length is 1 for 1 byte data , 2 for 2 byte data, 3 for 4 byte data.
- **Access Breakpoint** - An access breakpoint is triggered when memory location at the breakpoint address is either read or written. Access breakpoints also have lengths similar to write breakpoints. IO breakpoints in ia-32 are not supported.
  Since GDB stub at present does not use the protocol used by GDB for hardware breakpoints, hardware breakpoints are accessed through GDB macros. GDB macros for hardware breakpoints are described below.
  hwebrk: - Places an execution breakpoint
  usage hwebrk breakpointno address
  hwwbrk - Places a write breakpoint
  usage: hwwbrk breakpointno length address
  hwabrk - Places an access breakpoint
  usage: hwabrk breakpointno length address
  hwrmbrk - Removes a breakpoint
  usage: hwrmbrk breakpointno
  exinfo - Tells whether a software or hardware breakpoint has occurred. Prints number of the hardware breakpoint if a hardware breakpoint has occurred.
  Arguments required by these commands are as follows
  breakpointno - 0 to 3
  length - 1 to 3
  address - Memory location in hex digits (without 0x) e.g c015e9bc

Debugging modules

Inline Functions

The procedure for debugging modules is somewhat different because module object files do not contain absolute addresses.
A module is relocated by insmod command before it is loaded into the kernel. Relocation of a module is changing relative references in the module to absolute addresses where the module will be loaded.

Unloading a module and loading it again

kgdb versions 1.9 and above are capable of detecting loading and unloading of modules. You can load and unload modules as and when needed. gdb will automatically load and unload object files corresponding to them.

Debugging *init_module*

A special procedure is needed to debug *init_module* function from a module. Module debugging information is available only when gdb is notified of the module. *init_module* function from a module is already executed by that time. Hence *init_module* cannot be debugged with this procedure.

To debug *init_module,* place a breakpoint in the kernel just before the point where it calls *init_module* function.

Load the module with modprobe or insmod and wait for the breakpoint to occur. When the breakpoint occurs gdb has already detected that the module was loaded. You can now place a breakpoint anywhere in the module.

Architecture Dependencies

Debugging on x86_64

This KGDB patch has been tested on a dual CPU opteron machine. All kgdb features including thread support and console messages through gdb were tested and found to work.

x86_64 gdb cannot show stack trace correctly beyond *do_IRQ* function on the CPU which runs kgdb. For getting this stack trace, a shadow thread has been implemented. This thread's stack trace is the missing stack trace. It's shown in the example below:

Press *Ctrl+C* in gdb

```
Program received signal SIGTRAP, Trace/breakpoint trap.
  breakpoint () at kernel/kgdbstub.c:1056
  1056 atomic_set(&kgdb_setting_breakpoint, 0);
  (gdb) bt
  #0 breakpoint () at kernel/kgdbstub.c:1056
  #1 0xffffffff801e3f17 in kgdb8250_interrupt (irq=3, dev_id=0x0, regs=0x1)
  at drivers/serial/kgdb_8250.c:143
  #2 0xffffffff80113341 in handle_IRQ_event (irq=3, regs=0x10001a1dc48,
  action=0x1001ff29840) at arch/x86_64/kernel/irq.c:219
  #3 0xffffffff801134e1 in do_IRQ (regs=0x10001a1dc48)
  at arch/x86_64/kernel/irq.c:387
  #4 0xffffffff80110eab in common_interrupt () at elfcore.h:92
  Previous frame inner to this frame (corrupt stack?)
```

gdb can't trace beyond this point correctly. Check whether thread listing contains a

shadow thread. It can be identified by tag Stack at interrupt entrypoint.

(gdb) info thr
10 Thread 32769 (Stack at interrupt entrypoint) __delay (loops=1384000)
at arch/x86_64/lib/delay.c:30
9 Thread 32768 (Shadow task 0 for pid 0) 0xffffffff8010eb3e in cpu_idle ()
at sched.h:914
8 Thread 8 (aio/0) 0xffffffff8013f41c in worker_thread (
__startup=0x1001ff28d60) at sched.h:914
7 Thread 7 (kswapd0) kswapd (p=0xffffffff803170e8) at mm/vmscan.c:1046
6 Thread 6 (pdflush) __pdflush (my_work=0x1001fd71f18) at current.h:11
5 Thread 5 (pdflush) __pdflush (my_work=0x1001fd73f18) at current.h:11
4 Thread 4 (kblockd/0) 0xffffffff8013f41c in worker_thread (
__startup=0x1001ff28ae0) at sched.h:914
3 Thread 3 (events/0) 0xffffffff8013f41c in worker_thread (
__startup=0x1001ff3cd60) at sched.h:914
2 Thread 2 (ksoftirqd/0) ksoftirqd (__bind_cpu=0xffffffff8030be20)
at current.h:11
** 1 Thread 1 (swapper) breakpoint () at kernel/kgdbstub.c:1056*

It is thread 10 in above listing. We can get the lost back trace from thread 10

(gdb) bt
#0 __delay (loops=1384000) at arch/x86_64/lib/delay.c:30
#1 0xffffffff80212bfd in ide_delay_50ms () at drivers/ide/ide.c:1451
#2 0xffffffff8020b252 in actual_try_to_identify (drive=0xffffffff803c05e8,
cmd=236 ' ') at drivers/ide/ide-probe.c:351
#3 0xffffffff8020b67a in try_to_identify (drive=0xffffffff803c05e8,
cmd=236 ' ') at drivers/ide/ide-probe.c:405
#4 0xffffffff8020b7ef in do_probe (drive=0xffffffff803c05e8, cmd=236 ')
at drivers/ide/ide-probe.c:497
#5 0xffffffff8020bcaa in probe_hwif (hwif=0xffffffff803c04a0)
at drivers/ide/ide-probe.c:613
#6 0xffffffff8020bf46 in probe_hwif_init (hwif=0xffffffff803c04a0)
at drivers/ide/ide-probe.c:868
#7 0xffffffff8021a10d in ide_setup_pci_device (dev=0x1f9e8, d=0x1f7)
at drivers/ide/setup- pci.c:740
....

- ## Debugging on PowerPC

  This support is provided on experimental basis only, It hasn't been tested yet, so use with care. Any problem reports and fixes are most welcome!

  - ### Compiling kernel with KGDB

    Enable Standard/Generic serial support. Enable support for console on a serial port if console message through GDB is selected in KGDB options

Using KGDB over Ethernet interface

Kgdb 2.3 and 2.4 (2.6.13 and 2.6.15.5 resp.) work fine over kgdb interface. For using kgdb over ethernet interface one has to enable the option in kernel hacking while configuring the kernel.
e.g

Kernel hacking ->
[*] KGDB: kernel debugging with remote gdb ->
[*] KGDB: Console messages through gdb
Method for KGDB communication (KGDB: On generic serial port (8250)) --->
( ) KGDB: Use only kernel modules for I/O
( ) KGDB: On generic serial port (8250)
(X) KGDB: On ethernet - in kernel

After this just add following lines in kernel command line : *kgdboe=@10.0.0.6/,@10.0.0.3/ (that's kgdboe=@LOCAL-IP/,@REMOTE-IP/)*
Sample grub.conf which will by default boot the kgdb enabled kernel

title Linux-2.6.15.5-kgdb(eth)
root (hd0,0)
kernel /boot/bzImage- 2.6.15.5-kgdb ro root=/dev/hda1 kgdboe=@10.0.0.6/,@10.0.0.3/ console=ttyS0,115200

Then for starting the debug session give following command on gdb. *(gdb) ./vmlinux*
(gdb) target remote udp:HOSTNAME:6443

Troubleshooting

- ## Connection Problems

  gdb prints following errors on giving command target remote.

  Ingore packet error, continuing...
  Ingore packet error, continuing...
  Ingore packet error, continuing...

Couldn't establish connection to remote target
Malform response to offset query, timeout.

- Check whether the serial line speed given to the test kernel from lilo.conf or grub.cfg file on the test machine is same as the serial line speed on the development machine. They should be equal.
- Check whether the serial line is working properly. Boot the test machine in a non-debug kernel and check whether characters sent into the serial line from either end of the serial line appear on the other end. This can be done as follows:
  - Boot the test machine with the kernel that came with redhat installation. On the development machine run cat < /dev/ttyS0. It will wait for some characters to come from the serial line.
  - On the test machine run echo hello> /dev/ttyS0.
  - The cat running development machine machine should show "hello". There could be some extra end of lines. Ignore them.
  - Now kill the cat and repeat the same procedure with cat on the test machine and echo on development machine. This time the cat on test machine should show "hello".
    If either of the machines don't show hello you know that characters are not being sent. Check the serial line to see whether you have wired it incorrectly if that is the case.
- Check whether the prompt sent by kgdb is received on the development machine. This can be done as follows:
  - Run minicom on the serial line on the development machine. Setup serial line baud rate to appropriate value.
  - Now boot the test kernel.
  - When the message Waiting for connection from remote gdb... appears on the console of the test machine, following characters should be seen in minicom. +$S05#b8. This character string may repeat. It is the prompt sent to gdb from kgdb. If you don't see this serial line speed is set incorrectly on one of the machines.

• Problems with Breakpoints

- A breakpoint does not get hit as expected: Check whether you are using appropriate vmlinux file. Print the address of the function sys_close from gdb:
  (gdb) p sys_close
  $1 = {long int (unsigned int)} 0xc013212c <sys_close>
- Print address of the same function using proc filesystem on the test machine.
  $ grep sys_close /proc/ksyms
  c013212c sys_close_Rsmp_268cc6a2

The address is not the same, c013212c in the above case, vmlinux file being used is incorrect.

Frequently Asked Questions

After i apply the KGDB patches to kernel, complie and boot with the new image, my machine appears to hang up. Whats wrong?

KGDB for Linux 2.6.8-rc1 onwards uses early_param to pass the control to KGDB code in the kernel during the bootup. This is done very early in the kernel boot up before any messages can be enabled. Hence during the boot up the kernel is actually waiting for connection from remote GDB but since it cannot display any messages it appears that the machine has hung.

Why are two machines necessary for KGDB?

KGDB requires gdb for handling source code and debug information generated by gcc. gdb cannot be run on the test machine when the kernel is in debug state. Hence gdb has to be run on a separate machine which contains a running kernel.

Can I place breakpoints in interrupt handlers?

Yes, you can. Breakpoints can be placed almost anywhere in a kernel. KGDB cannot handle breakpoints only in the parts of the kernel which are used by it (KGDB). These include the KGDB serial line handler and interprocessor interrupt handlers.

Why does a kernel and modules need to be compiled on a development machine instead of a test machine?

gdb needs to refer to source code files and vmlinux or a module object file. Since gdb runs on the development machine, these files are required to be present on the development machine. The test machine needs vmlinuz or module object files only. If a kernel or modules are compiled on the development machine, only vmlinuz or module object files need to be copied to the test machine. On the other hand, compiling a kernel or modules on the test machine makes it it necessary to copy object files as well as source code files to the development machine. Hence a development model is much simplified if compilation is done on a development machine.

I am not able to debug my modules ...

For module debugging the appropriate GDB should be used on the development machine. Please see the module debugging page for more details.