GDB Cookbook

Jacqueline DeLorie
CS 502 Operating Systems

While working on the projects for this course students may encounter crashes within their Virtual Machine. This document will show you how to use GDB on the Virtual Machine provided by Professor Lauer. GDB is the GNU debugging tool, which allows the user to view the state of the program at various points allowing the user to assess and repair the given program. After reading this, the user will be able to use GDB to debug both user and kernel space programs.

Using GDB

The Virtual Machine that Professor Lauer supplied comes with GDB pre-installed. Therefore students should take advantage of it in order to debug their projects.

There are two different ways to debug programs using GDB:

- One way is to use GDB while a program is running so that the user can see what is going on within the program itself.
- Another way to use GDB is to run it after the program has crashed, which can give you information about the state of the program at the time of the crash. This document will focus on using GDB on a running program, though most of the commands are the same for both.

Previous versions of the kernel did not have built in support for GDB. Different patches would be required depending on the version. Luckily for us, the version of Open Suse that Professor Lauer has provided does not require any of these patches.

GDB Setup

User Space Programs

The GDB setup for user space programs is simple; one terminal is used to run the program and another is used to run GDB. The only extra step is during compilation, the user must compile the program with the --g argument specified. The --g flag tells the compiler to store debugging information in the generated object file.

Kernel Debugging

Setting up GDB for kernel debugging is more involved. Two Virtual Machines must be used, one to run the kernel and another to run GDB. From here on the Virtual Machine that will run GDB will be referred to as Machine A and the Virtual Machine running the kernel will be referred to as Machine B.

The first step to kernel debugging with GDB is to enable KGDB inside the kernel you will be testing. KGDB flags the kernel to allow for debugging by GDB.

Start with the normal building process:

1. cd kernelSrc
2. make O=~/kernel-dest xconfig
3. Load the config Professor Lauer provided.
4. Once in the kernel configuration menu, change the name of the kernel.

a. Click “General setup”.
b. To the right double click and edit “Local version”.

5. KGDB must now be enabled.
   a. Choose “Kernel Hacking” from the main menu.
   b. Check “KGDB:kernel debugger”. To the right you will see that “KGDB: use kgdb over the serial console” has been automatically checked.
   c. Save the configuration. See Figure 1 below for an example.
6. make –j4 O=~/kernel-dest > ~/make-out.txt 2> ~/make-errors.txt
7. cd ~/kernel-dest
8. sudo make modules_install install

Note: As in user space programs the compiler must be told to generate debug information for GDB to use. This configuration option is CONFIG_DEBUG_INFO. We do not need to enable this as Professor Lauer’s configuration already has done so.

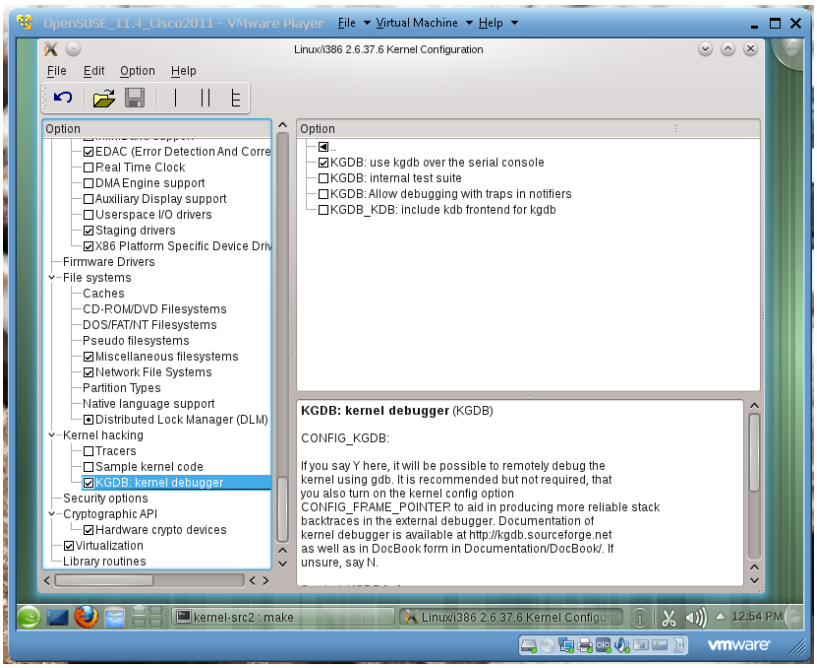

Figure 1

A second Virtual Machine will be needed after this point. Therefore, follow the instructions provided by Professor Lauer to create Machine B. Here is a link to the .pdf- SettingUpYourVirtualMachine_VMware.pdf

Machine A setup

Now that you have both Machine A and B you will need to copy the kernel from Machine A to Machine B. The kernel files necessary to run the test to be debugged on Machine B must be obtained from Machine A.

Gather the files necessary for Machine B that are located on Machine A:

1. In your home directory make a subdirectory called “image”.
2. Copy the following files from /grub/ to ~/image/:
   - System.map-2.6.37.6-*kernel name*
   - initrd-2.6.37.6-*kernel name*
   - vmlinuz-2.6.37.6-*kernel name*
3. Compress the folder from your home directory by typing “tar-cf image.tar image”.

4. Send image.tar to Machine B

Note: You could use the Shared Folder to do this or you could e-mail it to yourself. Remember that when you are not booted into the kernel supplied by Professor Lauer that the Shared Folder will not work.

Machine A and B must have the ability to communicate in a bidirectional fashion. Therefore, a serial connection is required between the two Virtual Machines. A socket must be opened up on each end of the connection. In order to create a serial connection, the Virtual Machine Settings must be edited.

To accomplish this:

1. Turn the Virtual Machine off.
2. Click "Edit virtual machine settings", then the "Add…" button; this will start the "Add Hardware Wizard".
3. Choose the "Serial Port" option, as seen in Figure 2.

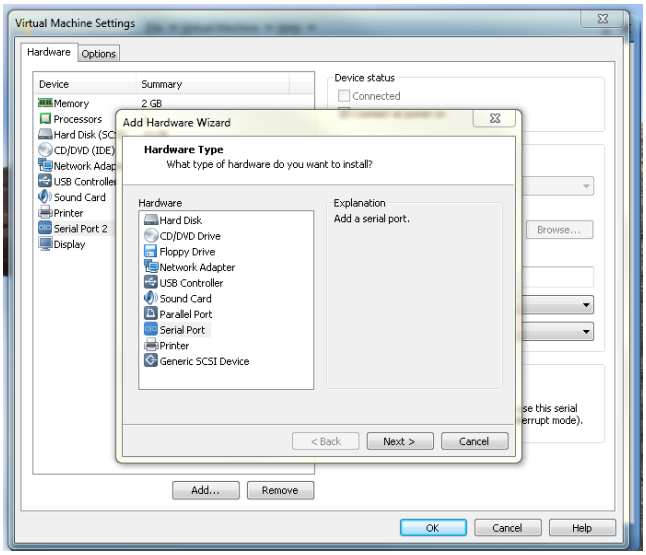

Figure 2

4. Click the "Next" button, then "Output to Named Pipe". Verify that the input information looks like Figure 3.

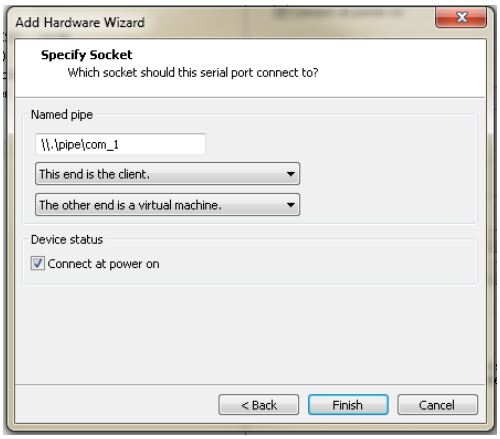

Figure 3

Machine B Setup

The image.tar file from Machine A must be un-tarred and put into the correct location:

1. Use "tar –xf image.tar".
2. Copy all of the contents from image to /boot.
3. Add the kernel entry to the grub list:
    a. In /boot/grub "sudo vim menu.lst".
    b. Add another entry for the kernel image you copied, use Figure 4 as an example.

Note: Two more boot arguments are required in order for Machine B to be controlled by GDB on boot. Add to the entry you just created "kgdb=ttyS1,115200 kgdbwait", see Figure 4 below. The argument "kgdb=ttyS1,115200" tells the kernel which socket it will be using to communicate with GDB and at what speed. "kgdbwait" tells the kernel to wait for a GDB connection.

Note: During boot "kgdbwait" will make the kernel hang until Machine B has executed the "continue" command.

Note: The number in "ttyS*number*" depends on how many serial ports are already in use. Within the given Virtual Machine image ttyS0 is in use by default. ttyS1 will work for this Virtual Machine, but if you have previously added any other serial connections you may need to increase this number.

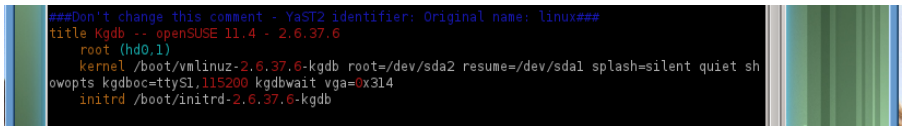

Figure 4

As in Machine A, a socket must be opened, as previously described above in Section "Machine A Setup". The one difference is in Step 4. When creating the socket, the side located on Machine B will be the server end. See this difference reflected in Figure 5.

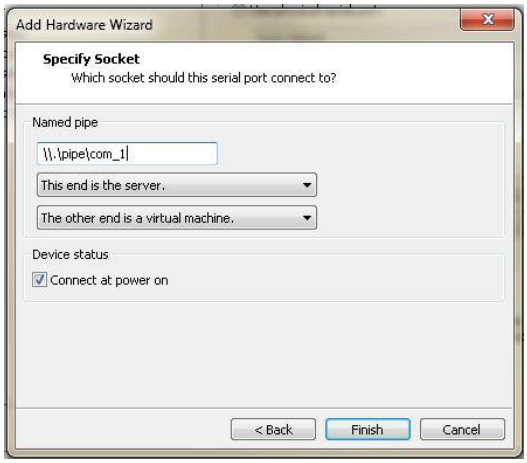

Figure 5

Note: In the client/server relationship within the socket, Machine B acts as the server pushing status information out to the client, Machine A.

At this point both Virtual Machines should be off. Machine B must be started first because it has been configured to have the server end of the connection. When Machine B is started, it will open the socket in order for Machine A to establish a connection. Now GDB can be launched on machine A and control it the same as it would for a user space program.

Using GDB during Program Execution

Starting GDB and Running Program through GDB

Here are different ways to invoke GDB to debug a running program:

1. gdb – This will start gdb up, but will not set up or start a program.
2. gdb *program name* - This will start gdb up and set the executable to you program's name.
3. gdb --args *program name* *arguments*- Will start gdb, set the executable to the given program name along with setting the arguments that will be passed into your program.

User Space Specific

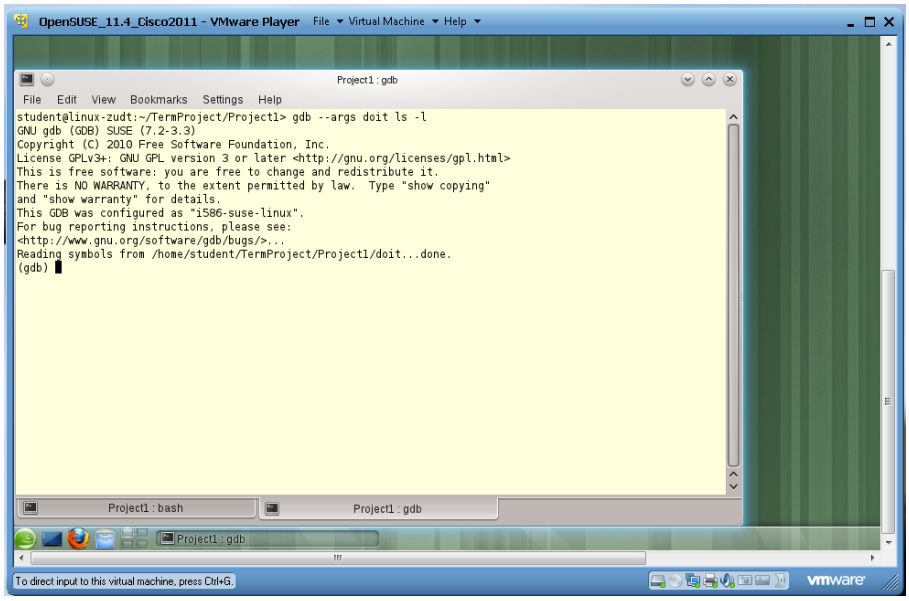

Figure 6

Once any of the above commands are executed the output shown in Figure 6 will be displayed. In order to run the specified program use the "run" command. If you forget to specify the program to be executed or wish to change it, the "file" command can be used. The use of this command is just "file *program name*". Also, if you forgot to specify the command line arguments you can use the "run" command to do so. This can also include the redirection of program output, which otherwise will display on-screen as normal. The format of the run command will then turn into "run *command line arguments*".

Kernel Specific

Machine B is already running the kernel that you will be debugging and is currently in the waiting state. There will be no need to tell the program to "run", but you do need to give GDB symbol information about the kernel. The symbol information allows you to gather accurate information along with control of the kernel. GDB will also need to be told what socket to listen to. The steps are below along with Figure 7 for directions.

Steps to start GDB on Machine A:

1. cd ~/kernel-dest
2. sudo gdb vmlinux (this load takes ~1 minute)
3. set remotebaud 115200
4. target remote /dev/ttyS1

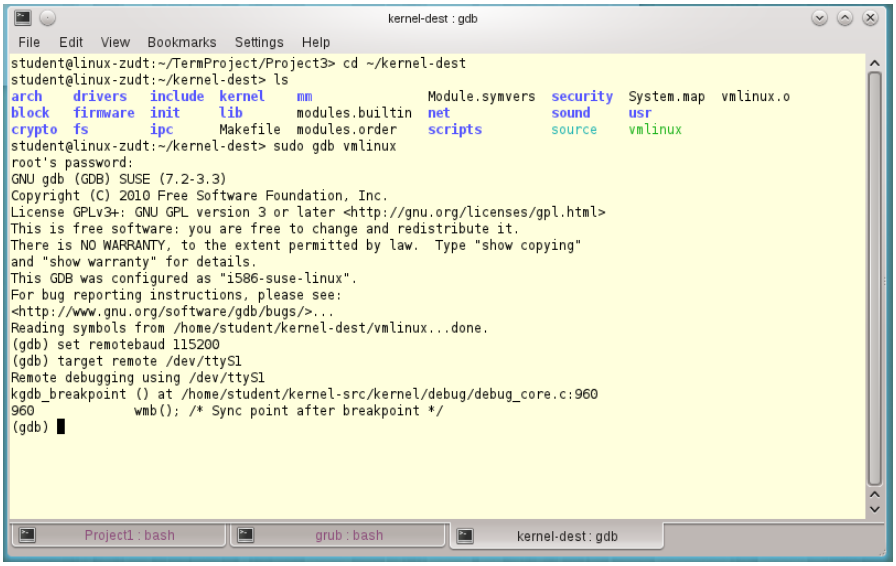

Figure 7

Multithreaded Programs:

GDB works with multithreaded programs, though only one thread is at the forefront at a time. The thread at the forefront is considered the "current" thread. Some commands can be executed on a specific thread or set of threads, but when a command is executed without a thread specified then the "current" is affected. To execute a command on all threads you can use "thread apply all *command*". Information is displayed whenever a thread is created or exited. To invoke the display of all threads use "info threads" or to switch the "current" thread use "thread *number*".

To view the current threads running on Machine A type "info threads" and you will see a list like the one shown in Figure 8.

Note: that Thread 1 is the "current" thread, which is distinguished by the "*" next to the number.

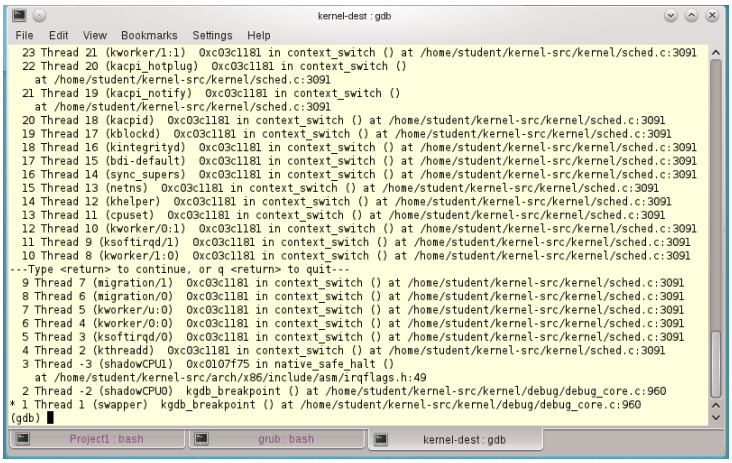

Figure 8

Breakpoints

Breakpoints are used to stop your program whenever a specific place in the code is hit. Breakpoints can be used to gather information about the state of the program at that point during execution. These can be particularly helpful for debugging crashes, if the user knows where in the code the program is crashing. Then the user can both stop the program (or Virtual Machine) from crashing and obtain information about the cause of the crash.

Breakpoint command:

break *location* thread *thread number* if *condition*

All arguments of the command are optional. If no arguments are specified the current line of execution will be used as a breakpoint.
The location can be:

1. Line number –break example.c:123
2. Function – break example.c:my_example_function
3. Address – break 0xffffffff

Thread number can be any of the thread id's listed by "info threads". If the "if condition" is not supplied then the breakpoint will always be enforced, otherwise you may supply a conditional statement in C. See Figure 9 below for an example of a breakpoint using the conditional statement to stop breaking after the fourth hit.

```
(gdb) set $foo = 3
(gdb) break execvp if $foo-- >= 0
```

Figure 9

In Machine A you will want to set a breakpoint before entering "continue", otherwise you will never be able to gain control of the kernel again. When debugging a problem within a custom kernel the output from /bin/dmesg, generated by an "Oops", can be helpful. By looking at these logs one can determine the point of the crash and set a break point before this line in the code.

Gathering Data

Once a breakpoint has been hit, information about the current state of the program can be gathered. The first step in this would be to print out the current stack, which are called "frames". To print the stack type "bt", also known as back trace. The local variables within the frames can be printed as well by adding "full", so "bt full". An example of "bt" output is show below in Figure 10. The stack trace can be helpful

when determining the code path followed when executed. You can change the "current" frame with "frame *number*", where number is one of the numbers listed by "bt".

```
(gdb) bt
#0  kgdb_breakpoint () at /home/student/kernel-src/kernel/debug/debug_core.c:960
#1  0xc016dc98 in kgdb_register_io_module (new_dbg_io_ops=0xc06470c8)
    at /home/student/kernel-src/kernel/debug/debug_core.c:900
#2  0xc0323834 in configure_kgdboc () at /home/student/kernel-src/drivers/serial/kgdboc.c:196
#3  0xc0101225 in do_one_initcall (fn=0xc0678d19 <init_kgdboc>) at /home/student/kernel-src/init/main.c:750
#4  0xc06528d9 in do_initcalls (unused=<value optimized out>) at /home/student/kernel-src/init/main.c:780
#5  do_basic_setup (unused=<value optimized out>) at /home/student/kernel-src/init/main.c:801
#6  kernel_init (unused=<value optimized out>) at /home/student/kernel-src/init/main.c:892
#7  0xc0102e26 in ?? () at /home/student/kernel-src/arch/x86/kernel/entry_32.S:1044
#8  0x00000000 in ?? ()
```

Figure 10

If you would like to see the code surrounding a certain line, then you can "list *filename:line number*". "List" alone will print the code surrounding the current instruction pointer within that frame. See Figure 11 below, which uses "list" to show a range in a file from the back trace in Figure 10. "List" is helpful while examining the back trace. Viewing the code is useful when traceing back where an argument was passed from or where a variable was modified.

```
(gdb) list debug_core.c:960,980
960             wmb(); /* Sync point after breakpoint */
961             atomic_dec(&kgdb_setting_breakpoint);
962     }
963     EXPORT_SYMBOL_GPL(kgdb_breakpoint);
964
965     static int __init opt_kgdb_wait(char *str)
966     {
967             kgdb_break_asap = 1;
968
969             kdb_init(KDB_INIT_EARLY);
970             if (kgdb_io_module_registered)
971                     kgdb_initial_breakpoint();
972
973             return 0;
974     }
975
976     early_param("kgdbwait", opt_kgdb_wait);
(gdb)
```

Figure 11

While inside a frame, more information about the current state of variables, registers, etc can be found. To show the current registers, "info registers" can be used. One register to note is "eip", this is the current instruction pointer. It is also the point in the code where "list" will display around if no other location is passed in. "info args" can be used to show the values of the arguments passed into the function. "info locals" will display the values of all of the local variables. Figure 12, below, shows an example for all three of the previous commands. Showing the values of these variables allows the user to see if any unexpected values are occurring. Unexpected variable values could change the code path or cause crashes. Variables such as function pointers could drastically change the code path if accidentally altered. GDB allows the user to view the function pointer address and then view the function pointed to by that address. This is done by dereferencing the pointer, which is described below.

```
(gdb) info args
No arguments.
(gdb) info locals
p = 0xf6c73ec0
tty_line = 1
err = -19
cptr = <value optimized out>
cons = 0x0
(gdb) info registers
eax            0x36     54
ecx            0x46     70
edx            0x46     70
ebx            0x0      0
esp            0xf409bf8c       0xf409bf8c
ebp            0xc0678d19       0xc0678d19
esi            0xf6c73ec0       -154714432
edi            0x80     128
eip            0xc0323834       0xc0323834 <configure_kgdboc+241>
eflags         0x202    [ IF ]
cs             0x60     96
ss             0x68     104
ds             0xf6c7007b       -154730373
es             0x7b     123
fs             0xffff   65535
gs             0xffff   65535
(gdb)
```

Figure 12

In the above example, seen in Figure 12, you can see that info locals printed an address as the value for p. P is actually a pointer therefore, the pointer's memory address is being displayed rather than the value of the structure. The values within the structure can be seen by dereferencing the pointer; the same as is done in C, see Figure 13 below for an example. If the value of the pointer is 0x0, then it is null. If the value within the structure is referenced while the pointer is still null, then a null pointer error will occur, also known as a segmentation fault. Knowing the value of a variable can also be helpful when an assert statement is failed and an error is thrown for that reason.

```
(gdb) print p
$4 = (struct tty_driver *) 0xf6c73ec0
(gdb) print *p
$5 = {
  magic = 21506,
  kref = {
    refcount = {
      counter = 2
    }
  },
  cdev = {
    kobj = {
      name = 0x0,
      entry = {
        next = 0xf6c73ecc,
        prev = 0xf6c73ecc
      },
      parent = 0x0,
      kset = 0x0,
      ktype = 0xc06397e0,
      sd = 0x0,
      kref = {
        refcount = {
          counter = 1
        }
      },
      state_initialized = 1,
      state_in_sysfs = 0,
```

Figure 13

Summary

GDB is an extremely useful tool for both user and kernel space programs. You should now be able to set up and start GDB on both user and kernel space programs. Once GDB has started you should be able to set breakpoints and navigate your way around the call stack of various threads when these breakpoints are hit. GDB has much more functionality than would ever be able to fit into this cookbook, but you should now be able to successfully find and address segmentation fault issues. Attached is a "cheat sheet" of the above commands for quick reference.

Cheat Sheet

gdb – This will start gdb up, but will not set up or start a program.
gdb *your program name* - This will start gdb up and set the executable to you program's name.
gdb --args *your program name* *arguments*- Will start gdb, set the executable to your programs name. along with setting the arguments that will be passed into your program.
file *program* - set the executable/file.
run – run the set executable with the currently set arguments.
run *arugments* - run the set executable with the given arguments
info threads – lists the currently running threads (* next to the current thread)
thread apply all *command* - apply command to all threads
thread *number* - switch thread to given thread number
break *location* thread *thread number* if *condition* - set a break point at the given location, thread number, only when the given condition is true(by default the if is always true)
bt – dump the current call stack
frame *number* – change to given frame number
list – lists code lines around the current instruction pointer
list *filename:line number* -list code lines around the given line
info registers – dump registers
info args – display arguments passed to the function
info locals – display the local variables
print *pointer – print the structure pointed to by pointer

Bibliography

"Debugging with GDB." *Sourceware.org: Free Software! Get Your Fresh Hot Free Software!* Web. 09 Jan. 2012. <http://sourceware.org/gdb/current/onlinedocs/gdb/index.html>.

Web. 09 Jan. 2012. <http://speed.cis.nctu.edu.tw/~ydlin/course/cn/mcn10fg/KGDB_HOWTO.pdf>.