CS 161: Operating Systems

Tuesday/Thursday 1:00-2:30

Pierce 301

What is GDB and why you need to use it

GDB is a debugger. Its purpose is to help you analyze the behavior of your program, and thus help you diagnose bugs or mistakes. With GDB you can do the following things:

- Control aspects of the environment that your program will run in.
- Start your program, or connect up to an already-started copy.
- Make your program stop for inspection or under specified conditions.
- Step through your program one line at a time, or one machine instruction at a time.
- Inspect the state of your program once it has stopped.
- Change the state of your program and then allow it to resume execution.

In your previous programming experience, you may have managed without using a debugger. You might have been able to find the mistakes in your programs by printing things on the screen or simply reading through your code. Beware, however, that OS/161 is a large and complex body of code, much more so than you may have worked on in the past. To make matters worse, much of it was written by someone other than you. A debugger is an essential tool in this environment. We would not lie if we said that there has never been a student in CS161 who has survived this class without using GDB. You should, therefore, take the time to learn GDB and make it your best friend (or rather your second best friend; your best friend should be your partner). This guide will explain to you how to get started debugging OS/161, describe the most common GDB commands, and suggest some helpful debugging techniques.

How to start debugging

OS/161 To debug OS/161, you should use the CS161 version of GDB, which is accessible as os161-gdb. This copy of GDB has been configured for the MIPS architecture and has been patched to be able to talk to System/161. The difference between debugging a regular program and debugging an OS/161 kernel is that the kernel is running in a machine simulator. You want to debug the kernel; running the debugger on the machine simulator is not very illuminating and we hope it will not be necessary. If you were to type:

```
% os161-gdb system161
```

you would be attempting to debug the simulator. This will not work, because the simulator is not compiled for MIPS. (If you do need to debug the simulator at some point, you would use the regular system copy of GDB.) So you must type this:

```
% os161-gdb kernel
```

You will find, however, that having done this, telling GDB to run the kernel does not work, because the kernel has to be run on System/161.

Instead, what you need to do is start your kernel running in System/161; then run os161-gdb on the same kernel and tell it to attach to the copy you started running. To do this you have to tell GDB to talk to System/161's debugger port.

This requires two windows, one to run the kernel in and one to run GDB in. These two windows must be logged into the same machine. It will not work if they are not. Fortunately, since you're running in your own virtual machine, if you open two windows, they are running on the same machine. (However, if you ever find yourself debugging kernels in some other environment, these few tidbits of information might come in handy!)

Now you are ready to debug. In one window (the run window), boot OS/161 on System/161. Use the -w option to tell System/161 to wait for a debugger connection:

```
% cd ~/cs161/root
% sys161 -w kernel
```

Next, in the other window (your debug window), run os161-gdb on the same kernel (if you run it on a different kernel by accident, you'll get bizarre results) and tell GDB to connect to System/161:

```
% cd ~/cs161/root
% os161-gdb kernel
(gdb) target remote unix:.sockets/gdb
```

GDB will connect up and it will tell you that the program is stopped somewhere in start.S. It is waiting at the very first instruction of your kernel, as if you'd run it from GDB and put a breakpoint there. Unfortunately, the directory in which you are running (`cs161/root`) is not the one in which you built your kernel, so the pathnames in the kernel executable aren't quite correct. Fortunately, you can fix that. You just have to tell `gdb` the directory from which to resolve path names. Type:

```
(gdb) directory /home/jharvard/cs161/os161/kern/compile/ASSTN
```

where N is the number of the assignment you are debugging, e.g., ASST0, ASST1, etc.

Now, you can use GDB to debug as you would debug any other program. We'll give you some practice later. When you are done debugging, you can disconnect the debugger from System/161 (and thus the running kernel) using the detach command:

```
(gdb) detach
```

You can also, instead, tell GDB to kill the process it's debugging. This will cause System/161 to exit unceremoniously, much as if you'd gone to its window and typed ^C:

```
(gdb) kill
```

Note that you do not necessarily need to attach GDB to System/161 at startup. You can attach it at any time. However, for reasons we do not presently understand, connecting does not always work properly unless System/161 is stopped waiting for a debugger connection. You can put it into this state at any time by typing ^G into its window. This can be useful if your kernel is looping or deadlocked.

Most common GDB commands

l, list - List lines from source files. Use this command to display parts of your source file. For example, typing

```
(gdb) l 101
```

will display line 101 in your source file. If you have more than one source file, precede the line number by the file name and a colon:

```
(gdb) l os.c:101
```

Instead of specifying a line number, you can give a function name, in which case the listing will begin at the top of that function.

b, break - set a breakpoint. Use this command to specify that your program should stop execution at a certain line or function. Typing

```
(gdb) b 18
```

means that your program will stop every time it executes a statement on line 18. As with the "list" command, you can specify to break at a function, e.g.:

```
(gdb) b main
```

d, delete - Delete breakpoint (or other things). Use this command to delete a breakpoint. By typing

```
(gdb) d 1
```

you will delete the breakpoint number "1". GDB displays the number of a breakpoint when you set that breakpoint. Typing "d" without arguments will cause the deletion of all breakpoints.

clear - Clear a breakpoint. If you don't remember the number of the breakpoint you want to delete, use the "clear" command. Just like the "breakpoint" command, it takes a line number or a function name as an argument.

c, continue - continue execution. After your program has stopped at a breakpoint, type

```
(gdb) c
```

if you want your program to continue the execution until the next breakpoint.

s, step - Step through the program. If you want to go through your program step by step after it has hit a breakpoint, use the "step" command. Typing

```
(gdb) s
```

will execute the next line of code. If the next line is a function call, the debugger will step into this function.

n, next - Execute the next line. This command is similar to the "step" command, except for it does not step into a function, but executes it, as if it were a simple statement.

disable, enable - Disable/enable a breakpoint. Use these commands with a breakpoint number as an argument to disable or enable a breakpoint.

display - Display an expression. Display a value of an expression every time the program stops. Typing

```
(gdb) display x
```

Will print the value of a variable "x" every time the program hits a breakpoint. If you want to print the value in hex, type:

```
(gdb) display /x x
```

undisp - Cancel the display of some expressions. Arguments are the code numbers of the expressions. If no arguments are given, GDB will cancel all expression displays.

print, printf - Print an expression. To print a value of an expression only once, use the "print" command. It takes the same arguments as the "display" command.

The "printf" command allows you to specify the formatting of the output, just like you do with a C library printf() function. For example, you can type:

```
(gdb) printf "X = %d, Y = %d\n",X,Y
```

command - Execute a command on a breakpoint. You can specify that a certain command, or a number of commands be executed at a breakpoint. For example, to specify that a certain string and a certain value are printed every time you stop at breakpoint 2, you could type:

```
(gdb) command 2
 > printf "theString = %s\n", theString
 > print /x x
 > end
```

bt, where - Display backtrace. To find out where you are in the execution of your program and how you got there, use one of these commands. This will show the backtrace of the execution, including the function names and arguments.

set - Assign a value to a variable. Sometimes it is useful to change the value of a variable while the program is running. For example if you have a variable "x", typing

```
(gdb) set variable x = 15
```

will change the value of "x" to 15.

define - Define a new command. Use this if you want to define a new command. This is similar to assigning a macro and can save you typing. Just as with a macro, you can put together several commands. For example, if you were tired of typing

```
(gdb) target remote unix :.sockets/gdb
```

all the time, you could do:

```
(gdb) define db
Type commands for definition of "db".
End with a line saying just "end".
> target remote unix :.sockets/gdb
> end
```

Then you could invoke it just by typing "db". (If you put this or other commands in a file called .gdbinit, GDB will execute them automatically at startup time.)

info - Display information. With this command you can get information about various things in your

debugging session. For example, to list all breakpoints, type:

```
(gdb) info breakpoints
```

To see the current state of the hardware machine registers, type:

```
(gdb) info registers
```

help - Get help (on gdb, not your CS161 assignments!) Finally, if you want to find more about a particular command just type:

```
(gdb) help
```

Debugging tips

Tip #1: Check your beliefs about the program

So how do you actually approach debugging? When you have a bug in a program, it means that you have a particular belief about how your program should behave, and somewhere in the program this belief is violated. For example, you may believe that a certain variable should always be 0 when you start a "for" loop, or a particular pointer can never be NULL in a certain "if statement". To check such beliefs, set a breakpoint in the debugger at a line where you can check the validity of your belief. And when your program hits the breakpoint, ask the debugger to display the value of the variable in question.

Tip #2: Narrow down your search

If you have a situation where a variable does not have the value you expect, and you want to find a place where it is modified, instead of walking through the entire program line by line, you can check the value of the variable at several points in the program and narrow down the location of the misbehaving code.

Tip #3: Walk through your code

Steve Maguire (the author of Writing Solid Code) recommends using the debugger to step through every new line of code you write, at least once, in order to understand exactly what your code is doing. It helps you visually verify that your program is behaving more or less as intended. With judicious use, the step, next and finish commands can help you trace through complex code quickly and make it possible to examine key data structures as they are built.

Tip #4: Use good tools

Using GDB with a visual front-end can be very helpful. For example, using GDB inside the emacs editor puts you in a split-window mode, where in one of the windows you run your GDB session, and in the other window the GDB moves an arrow through the lines of your source file as they are executed. To use GDB through emacs do the following:

1. Start emacs.
2. Type the "meta" key followed by an "x".
3. At the prompt type "gdb". Emacs will display the message:

```
Run gdb (like this): gdb
```

4. Delete the word "gdb", and type:

```
os161-gdb kernel
```

So in the end you should have:

```
Run gdb (like this): os161-gdb kernel
```

displayed in the control window.

At this point you can continue using GDB as explained in section 2.

Tip #5: Beware of printfs!

A lot of programmers like to find mistakes in their programs by inserting "printf" statements that display the values of the variables. If you decide to resort to this technique, you have to keep in mind two things: First, because adding printfs requires a recompile, printf debugging may take longer overall than using a debugger.

More subtly, if you are debugging a multi-threaded program, such as a kernel, the order in which the instructions are executed depends on how your threads are scheduled, and some bugs may or may not manifest themselves under a particular execution scenario. Because printf outputs to the console, and the console in System/161 is a serial device that isn't extraordinarily fast, an extra call to printf may alter the timing and scheduling considerably. This can make bugs hide or appear to come and go, which makes your debugging job much more difficult.

To help address this problem, System/161 provides a simple debug output facility as part of its trace control device. One of the trace control device's registers, when written to, prints a notice in the System/161 output including the value that was written. In OS/161, provided your System/161 has been configured to include the trace control device, you can access this feature by calling trace_debug(), which is defined in dev/lamebus/ltrace.h. While this is less disruptive than calling printf, it is still not instant and can still alter the timing of execution. By contrast, the System/161 debugger interface is completely invisible; as far as your kernel is concerned, time is stopped while you are working in the debugger.

Tip #6: Debugging assembly

When GDB stops in an assembly source file (a .S file) various special considerations apply. GDB is meant to be a source-level debugger for high level languages and isn't very good at debugging assembly. So various tricks are required to get adequate results.

The OS/161 toolchain now tells the assembler to emit line number information for assembly files, so in theory you should at least be able to see the file you're working on. (If GDB can't find the file, you can use the path command to tell it where to look.)

It is also sometimes helpful to disassemble the kernel; type

```
% objdump --disassemble kernel | less
```

in another window and page or search through it as needed.

To single step through assembler, use the nexti and stepi commands, which are like next and step but move by one instruction at a time.

The command x /i (examine as instructions) is useful for disassembling regions from inside GDB.

Use the command info registers to see the values that are being handled. Unfortunately, you can't print only one register.

Tip #7: trace161

The trace161 program is the same as sys161 but includes additional support for various kinds of tracing and debugging operations. You can have System/161 report disk I/O, interrupts, exceptions, or whatever. See the System/161 documentation for more information.

One of the perhaps more interesting trace options is to have System/161 report every machine instruction that is executed, either at user level, at kernel level, or both. Because this setting generates vast volumes of output, it's generally not a good idea to turn it on from the command line. (It is sometimes useful, however, in the early stages of debugging assignment 2 or 3, to log all user-mode instructions.) However, the trace options can be turned on and off under software control using the System/161 trace control device. It can be extremely useful to turn instruction logging on for short intervals in places you suspect something strange is happening. See dev/lamebus/ltrace.h for further information.

Tip #8: Other tricks and caveats

If you have a void * in GDB and you know what type it actually is, you can cast it when printing, using the usual C expression syntax. This can be surprisingly useful; for instance, if you want to know what a thread is sleeping on, you can cast its sleepaddr pointer to struct lock *. This will tell you the name of the lock. (If the object is not a lock but a semaphore or CV, it still works, as long as the name field of those structures is in the same place as the name field of the lock.)

Another trick with a stopped thread is to cast thread->pcb.pcb_savestack to struct switchframe *; this will let you inspect its saved register values.

When you get a stack backtrace and it reaches an exception frame, GDB can sometimes now trace through the exception frame, but it doesn't always work very well. Sometimes it only gets one function past the exception, and sometimes it skips one function. (This is a result of properties of the MIPS architecture and the way GDB is implemented and doesn't appear readily fixable.) Always check the tf_epc field of the trap frame to see exactly where the exception happened, and if in doubt, cross-check it against a disassembly or have GDB disassemble the address.

Where to go for help

For help on GDB commands, type "help" from inside GDB. You can find the documentation on GDB here And of course your friendly TFs are always there to help!