COMP3231/9201/3891/9283 Operating Systems 2019/T1 UNSW

Administration
- Notices
- Course Outline
- UNSW Timetable
- Consultations
- Group Nomination
- Survey Results!!

Work
- Lectures
- Tutorials
- Extended Lectures

Support
- Piazza Forums
- Wiki

Assignments
- Assignment 0 Warm-up
- Assignment 1
- Assignment 2
- Assignment 3

Resources
OS/161
- General
- Man Pages
- Sys161 Pages

C coding
- Info Sheet

Debugging
- GDB and OS/161

General
- "Hardware" Guide
- R3000 Reference Manual
- Intro. to Prog. Threads

Previous years
- 2018 S1
- 2017 S1
- 2016 S1
- 2015 S1
- 2014 S1
- 2013 S1
- 2012 S1
- 2011 S1
- 2010 S1
- 2009 S1
- 2008 S1
- 2007 S1
- 2006 S1
- 2005 S2
- 2005 S1
- 2004 S2
- 2004 S1

Staff
- Kevin Elphinstone (LiC)
- Alex Kroh (Admin)

GDB and OS/161

This page contains a short tutorial on using GDB with OS161.

Setting up GDB

Every time you start GDB you will need to tell it the location of your source and how to communicate with System/161. This can become tedious, so we create a shortcut.

Place the following (adjusted for your setup, of course) into your root directory, usually `~/cs3231/root` in a file called `.gdbinit`.

```
set can-use-hw-watchpoints 0
define connect
dir ~/cs3231/asst0-src/kern/compile/ASST0
target remote unix:.sockets/gdb
b panic
end
```

Whenever you start GDB in this directory, you can type `connect` and the above commands will be run. Note that we also set a breakpoint at the `panic` function — whenever the kernel panics the debugger will be entered. Very useful.

Note: You may need to add the following to the separate file `~/.gdbinit`, if you see a warning regarding auto loading.

```
set auto-load safe-path /
```

An Example

Consider the following session based on a (hypothetical) buggy kernel. We wish to find the bug, so we start sys161 with the `-w` flag, which tells sys161 to wait for GDB. We get the following:

```
~/cs3231/root$ sys161 -w kernel
sys161: System/161 release 1.1, compiled Jul 28 2003 17:28:51
sys161: Waiting for debugger connection...
```

In another terminal, we have changed the directory to the root directory and run GDB. We run the `connect` command to setup GDB, and then we can let sys161 continue (the `c` command).

```
~/cs3231/root$ os161-gdb kernel
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=mips-elf"...
(gdb) connect
__start () at ../../arch/mips/mips/start.S:24
24      ../../arch/mips/mips/start.S: No such file or directory.
Current language:  auto; currently asm
Breakpoint 1 at 0x80010c94: file ../../lib/kprintf.c, line 94.
(gdb) c
Continuing.
```

In the terminal running sys161 we see the following output:

```
sys161: New debugger connection

OS/161 base system version 1.08
Copyright (c) 2000, 2001, 2002, 2003
   President and Fellows of Harvard College.  All rights reserved.

Put-your-group-name-here's system version 0 (ASST0 #3)

Cpu is MIPS r2000/r3000
344k physical memory available
Device probe...
lamebus0 (system main bus)
emu0 at lamebus0
ltrace0 at lamebus0
ltimer0 at lamebus0
hardclock on ltimer0 (100 hz)
beep0 at ltimer0
rtclock0 at ltimer0
```

Grievances - Student Reps

```
lrandom0 at lamebus0
random0 at lrandom0
lhd0 at lamebus0
lhd1 at lamebus0
lser0 at lamebus0
con0 at lser0
pseudorand0 (virtual)

panic: Fatal exception 3 (TLB miss on store) in kernel mode
panic: EPC 0x800197c4, exception vaddr 0x0
```

The kernel has hit a bug and panicked! The kernel tells us that it has taken a `TLB miss on store` in kernel mode, which means that some kernel code has written to an invalid address. The second line tells us that the address of the faulting instruction is `0x800197c4` and the address the instruction was trying to access is `0x0`.

Luckily, we set GDB to break whenever the `panic` function was invoked, so in our GDB terminal we see the following:

```
Breakpoint 1, panic (fmt=0x80023270 "I can't handle this... I think I'll just die now...\n") at ../../lib/kprintf.c:94
94              if (evil==0) {
Current language:  auto; currently c
```

At this point we would like to know where the exception occurred. There are two ways of doing this: using the `list` command or the `frame` command.

The `list` command is useful for finding the line of source that contains the faulting instruction (note the `*`):

```
(gdb) list *0x800197c4
0x800197c4 is in kmain (../../main/main.c:164).
159     kmain(char *arguments)
160     {
161         char *bad_ptr = NULL;
162         boot();
163
164         *bad_ptr = 0;
165
166         menu(arguments);
167
168         /* Should not get here */
```

In this case it is pretty easy to see what happened --- the `bad_ptr` variable was dereferenced without being initialised to a valid memory location.

Remember: this is a fictional example; your assignment 0 won't look the same.

If we wanted to examine the state of the program, we would use the `where` and `frame` commands:

```
(gdb) where
#0  panic (fmt=0x80023270 "I can't handle this... I think I'll just die now...\n") at ../../lib/kprintf.c:94
#1  0x8000cfbc in mips_trap (tf=0x80029f38) at ../../arch/mips/mips/trap.c:197
#2  0x8000bbb0 in common_exception () at ../../arch/mips/mips/exception.S:211
#3  0x800197c4 in kmain (arguments=0x80028af4 "") at ../../main/main.c:164
#4  0x8000c7e8 in __start () at ../../arch/mips/mips/start.S:163
```

We can see the functions that led to this exception: `__start` called `kmain` which took an exception, causing `common_exception` to be called and then so on down to `panic`.

We would like to know the line that caused the exception, and we would also like to examine the program's variables. To do this, we use the `frame` command, telling it that we would like to examine the frame associated with `kmain`, in this case frame 3 (the number next to kmain in the above output).

```
(gdb) frame 3
#3  0x800197c4 in kmain (arguments=0x80028af4 "") at ../../main/main.c:164
164         *bad_ptr = 0;
```

Looking at the above output, it seems that the store exception was caused by the store to `bad_ptr`. To make sure, we can get the value of `bad_ptr` using the `print` command (we can shorten this to `p`).

```
(gdb) p bad_ptr
$1 = 0x0
```

We have found the culprit! The kernel has tried to store to the memory pointed to by `bad_ptr`, however the location is invalid (0x0).

Watch Points

Sometimes bugs are not as obvious as in the previous example. We would like the ability to break into the debugger whenever a certain variable is modified. In GDB this is done with the `watch` command.

We start as before, debugging a buggy kernel:

```
(gdb) connect
__start () at ../../arch/mips/mips/start.S:24
24      ../../arch/mips/mips/start.S: No such file or directory.
Current language:  auto; currently asm
Warning: /import/paulaner/2/sjw/work/cs3231_03s2/source/os161-1.08/kern/compile/ASST0: No such file or directory.
Breakpoint 1 at 0x80010c94: file ../../lib/kprintf.c, line 94.
(gdb) c
Continuing.

Breakpoint 1, panic (fmt=0x800232c0 "I can't handle this... I think I'll just die now...\n") at ../../lib/kprintf.c:94
94              if (evil==0) {
Current language:  auto; currently c
(gdb) where
#0  panic (fmt=0x800232c0 "I can't handle this... I think I'll just die now...\n") at ../../lib/kprintf.c:94
#1  0x8000cfbc in mips_trap (tf=0x80029f40) at ../../arch/mips/mips/trap.c:197
#2  0x8000bbb0 in common_exception () at ../../arch/mips/mips/exception.S:211
#3  0x80019808 in kmain (arguments=0x80028b58 "") at ../../main/main.c:173
#4  0x8000c7e8 in __start () at ../../arch/mips/mips/start.S:163
(gdb) frame 3
#3  0x80019808 in kmain (arguments=0x80028b58 "") at ../../main/main.c:173
173             *good_ptr = 1;
(gdb) p good_ptr
$1 = (int *) 0x0
```

So far, so good. However, the cause of the bug isn't apparent. From looking at the code it seems that something is changing `good_ptr` between lines 171 and 173.

```
(gdb) list *0x80019808
0x80019808 is in kmain (../../main/main.c:173).
168     {
169             boot();
170
171             good_ptr = &some_int;
172             do_something();
173             *good_ptr = 1;
174
175             menu(arguments);
176
177             /* Should not get here */
```

We will use watchpoints to tell us when `good_ptr` changes. Unfortunately, watchpoints in system 161 can be very slow, so we would like to make sure that we only use watch points when we know something is going to break.

We start as before, but this time break at `kmain`, both to check the value of `good_ptr` and to set the watchpoint. Note the use of the `n`ext command to step over the uninteresting `boot()` function.

```
(gdb) break kmain
Breakpoint 5 at 0x800197d8: file ../../main/main.c, line 169.
(gdb) c
Continuing.

Breakpoint 5, kmain (arguments=0x80028b58 "") at ../../main/main.c:169
169             boot();
Current language:  auto; currently c
(gdb) n
171             good_ptr = &some_int;
(gdb) p good_ptr
$2 = (int *) 0x0
(gdb) s
172             do_something();
(gdb) p good_ptr
$3 = (int *) 0x80026c10
(gdb) p &some_int
$4 = (int *) 0x80026c10
```

At this point, `good_ptr` is as you would expect --- it points to `some_int`. We now know that something changes `good_ptr` in `do_something()`, so we set a watchpoint and continue.

```
(gdb) watch good_ptr
Watchpoint 6: good_ptr
(gdb) c
Continuing.
Watchpoint 6: good_ptr

Old value = (int *) 0x80026c10
New value = (int *) 0x0
do_something () at ../../main/main.c:160
160     }
```

Sure enough, the watchpoint has been tripped: the `do_something()` function has changed the value of `good_ptr` to `NULL`.

```
(gdb) list
155
156
```

```
157     static void do_something(void)
158     {
159             good_ptr = NULL;
160     }
161
162     /*
163      * Kernel main. Boot up, then fork the menu thread; wait for a reboot
164      * request, and then shut down.
```

Other Commands

Apart from the above commands, the following commands are useful:

- `step` tells GDB to execute one line of the program. If the line is a function call, the function is stepped into.
- `next` is similar to `step`, however functions are not followed --- the program runs until the control returns to the current function.
- `display` causes an expression (usually a variable) to be displayed every time you step --- useful for determining when a variable changes.
- ... see the GDB documentation for many more useful commands.

For those who prefer to use a GUI, try out ddd. However we provide no support for it.

```
% ddd -debugger os161-gdb kernel
```

Page last modified: 9:52pm on Saturday, 16th of February, 2019

Print Version