

Computer Science 161: Operating Systems

Section 0: Introduction to OS/161

CS161 Course Staff
cs161@fas.harvard.edu
<http://motelab.eecs.harvard.edu/bb/>

February 12, 2004

1 GDB

Sooner or later this semester you're going to need to use `gdb`. You'll probably be able to get by doing Assignments 0 and 1 without it, but it's surprisingly powerful and might save you a lot of time later.

1.1 Why not `kprintf()`?

- Your kernel is multi-threaded; it's hard to figure out what's going on from just print output.
- `kprintf()` is unwieldy—depending on what you're doing it might give you so much output that it'll be painful to slog through it
- Too slow! You're concerned with development time, not simulator time. For `printf()` debugging, you need to re-compile once every few minutes. Compiling OS/161 is surprisingly slow¹.
- It's really hard to inspect complicated data structures with `kprintf()`. It's easier to type `print *some_struct_name`

1.2 So how?

1.2.1 Initialization

You need to use two terminals. In the first terminal:

```
[0]ice3:~(1010)>cd ~/cs161/root
[0]ice3:~/cs161/root(1011)>sys161 -w kernel
sys161: System/161 release 1.11, compiled Feb  3 2004 17:33:22
sys161: Waiting for debugger connection...
```

Then, in the second:

```
[0]ice3:~/cs161/src/kern/compile/ASST5(1074)>cs161-gdb kernel
GNU gdb 6.0
Copyright 2003 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.
```

¹It turns out that it's slow because of NFS; try moving your files to `/tmp` (a local partition) on a workstation machine—it'll compile much faster, but you don't really want to do this.

```

This GDB was configured as "--host=i686-pc-linux-gnu --target=mips-elf"...
(gdb) target remote unix:../../../../root/.sockets/gdb
Remote debugging using unix:../../../../root/.sockets/gdb
__start () at ../../arch/mips/mips/start.S:24
24      addiu sp, sp, -20
Current language: auto; currently asm

```

1.2.2 Breakpoints

We continue with breakpoints. Breaking on `panic()` is *really* handy. `panic()` gets called on kernel faults and when assertions fail. (Another debugging hint: use a lot of assertions. You will be really glad you put in a lot of stupid assertions when some assertion from Assignment 1 fails when you're working on Assignment 3.) Then, you can type `bt` to get a backtrace, and use `up` and `down` to navigate up and down the call stack.

```

(gdb) b panic
Breakpoint 1 at 0x8001c6c8: file ../../lib/kprintf.c, line 94.
(gdb) c
Continuing.

```

[Type `panic` into first terminal]

```

Breakpoint 1, panic (fmt=0x80038f98 "User requested panic\n")
  at ../../lib/kprintf.c:94
94      DEBUG(DB_VM, "evil currently is: %d", evil);
Current language: auto; currently c
(gdb) bt
#0  panic (fmt=0x80038f98 "User requested panic\n") at ../../lib/kprintf.c:94
#1  0xffffffff80027344 in cmd_panic (nargs=1, args=0x8003eef4)
  at ../../main/menu.c:251
#2  0xffffffff800280e8 in cmd_dispatch (cmd=0x8003ef8c "panic")
  at ../../main/menu.c:761
#3  0xffffffff80028214 in menu_execute (line=0x8003ef8c "panic", isargs=0)
  at ../../main/menu.c:803
#4  0xffffffff800282d8 in menu (args=0x8003d5d0 "") at ../../main/menu.c:840
#5  0xffffffff80026cd8 in kmain (arguments=0x8003d5d0 "")
  at ../../main/main.c:190
#6  0xffffffff80017fe8 in __start () at ../../arch/mips/mips/start.S:163
(gdb) up
#1  0xffffffff80027344 in cmd_panic (nargs=1, args=0x8003eef4)
  at ../../main/menu.c:251
251      panic("User requested panic\n");
(gdb) up
#2  0xffffffff800280e8 in cmd_dispatch (cmd=0x8003ef8c "panic")
  at ../../main/menu.c:761
761      result = cmdtable[i].func(nargs, args);
(gdb) up
#3  0xffffffff80028214 in menu_execute (line=0x8003ef8c "panic", isargs=0)
  at ../../main/menu.c:803
803      result = cmd_dispatch(command);
(gdb) print *command
$1 = 112 'p'
(gdb) print command
$2 = 0x8003ef8c "panic"

```

1.2.3 Inspecting Memory

When you're working on, say, the `exec()` system call, you'll have to do a lot of careful pointer arithmetic. It turns out that it's really helpful to examine memory. Let's try it out with my implementation:

```
(gdb) l sys_exec
/* shows code for sys_exec() function */
(gdb) b 715
Breakpoint 3 at 0x8002b47c: file ../../userprog/proc_syscall.c, line 715.
(gdb) c
/* Type "s" to get a shell; then "/testbin/argtest cs161 rulez" */
Breakpoint 3, sys_exec (tf=0x80052f6c) at ../../userprog/proc_syscall.c:715
715         ret = copyargs_to_userland(exec_argbuf, &stackptr, totlen, nargs);
Current language: auto; currently c
(gdb) print (void *)stackptr
$4 = (void *) 0x80000000
(gdb) x /31c exec_argbuf
0x8003cdc0 <exec_argbuf_real>: 47 '/' 116 't' 101 'e' 115 's' 116 't' 98 'b' 105 'i' 110 'n'
0x8003cdc8 <exec_argbuf_real+8>: 47 '/' 97 'a' 114 'r' 103 'g' 116 't' 101 'e' 115 's' 116 't'
0x8003cdd0 <exec_argbuf_real+16>: 0 '\0' 99 'c' 115 's' 49 '1' 54 '6' 49 '1' 0 '\0' 114 'r'
0x8003cdd8 <exec_argbuf_real+24>: 117 'u' 108 'l' 101 'e' 122 'z' 0 '\0' 0 '\0' 0 '\0'
(gdb) n
(gdb) print (void *) stackptr
$7 = (void *) 0x7fffffd0
(gdb) x /50c stackptr
0x7fffffd0: 127 '\177' -1 '?' -1 '?' -30 '?' 127 '\177' -1 '?' -1 '?' -13 '?'
0x7fffffd8: 127 '\177' -1 '?' -1 '?' -7 '?' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fffffe0: 0 '\0' 0 '\0' 47 '/' 116 't' 101 'e' 115 's' 116 't' 98 'b'
0x7fffffe8: 105 'i' 110 'n' 47 '/' 97 'a' 114 'r' 103 'g' 116 't' 101 'e'
0x7fffff0: 115 's' 116 't' 0 '\0' 99 'c' 115 's' 49 '1' 54 '6' 49 '1'
0x7fffff8: 0 '\0' 114 'r' 117 'u' 108 'l' 101 'e' 122 'z' 0 '\0' 0 '\0'
0x80000000: 3 '\003' -96 '?'
```

1.2.4 More

There are lots of other useful `gdb` commands. Take a look at <http://www.courses.fas.harvard.edu/~cs161/handouts/gdb.html>.

1.2.5 Can I make it pretty?

It's worthwhile to fuss a bit to now and work on your development environment. Definitely use `.gdbinit` to alleviate common typing (like that `remote`) command. Write some shell scripts to set some stuff.

I setup `screen(1)` to give me a split-screen debugging view that simulated two windows in one. There are instructions at <http://www.courses.fas.harvard.edu/~cs161/handouts/screen.html>. It is conceivably possible to use `xxx` or `ddd` (X11 graphical debuggers) to debug with a graphical interface, but I don't know anyone who's done it. If you use the `emacs(1)` Operating System, there's probably a `gdb` mode in there somewhere.

If you do this stuff, share it with your classmates by posting to the bulletin board.

1.3 When to debug?

If you've set it up nicely, you can be in `gdb` all the time, with a breakpoint on `panic()`. Obviously you don't want to do performance analysis while debugging, but on the whole you might as well use it.²

²Be grateful you have a debugger. The Mica2 notes that Matt does research with have three LEDs to debug with which you're supposed to infer their state.

Some people say that you should step through every line of code that you write in the debugger. It's not a bad idea. You'll definitely want to step through tricky code.

1.4 One More Note

Don't be afraid of adding functionality to OS/161 if it seems worthwhile for debugging, even if it's not in the assignment. It's pretty easy to add a menu item to print some internal data out (or to make runtime changes). You could add a system call that lets user-level programs write integers via `kprintf()` (useful because one of you will be working on `exec()` while your partner works on `write()`).

2 The CS161 Toolchain

2.1 What is the CS161 toolchain?

The CS161 toolchain is the set of utilities (other than `sys161`) that you need to develop for the MIPS r2000/r3000. Again, we've got it compiled and built by default for you. If you want to work at home, you'll need to download and build this yourself.

2.2 Compilation/Debugging tools:

- `cs161-gcc` - the compiler for MIPS r2000/r3000. You should probably never have to invoke this directly.
- `cs161-gdb` - the debugger, which you will have to invoke more often than you care.
- `cs161-{ld, as, ar, ranlib, ...}` - all the normal build utilities you know and love.

2.3 Special Debugging Tools:

2.3.1 `stat161`

Description:

`stat161` displays useful statistics about how the various components of `sys161/os161`. This can be useful in profiling and detecting deadlocks.

Usage:

In order to use `stat161`, run the kernel and then run `stat161` from a different terminal session (make sure you're logged into the same box and are in the system root directory).

Interpreting `stat161` output:

- `kern` - cycles spent executing kernel code.
- `user` - cycles spent executing user level code.
- `idle` - idle cycles.
- `irqs` - interrupts raised.
- `exns` - exceptions.
- `disk` - disk usage.
- `con` - console usage.
- `emu` - emufs usage.

- `net` - network interface usage.

Identifying deadlocks:

If running a user-level program (e.g. user-level tests), check for user activity. Look for time spent in kernel, if this is about the same the entire time, that's usually a good sign of deadlock (≈ 33000 is the standard, if you're seeing just that, either your system is idle or there's a good chance you're in a deadlock). If thread uses disks, look for disk activity.

2.3.2 `cs161-objdump`

Description:

`cs161-objdump` can be used to disassemble user level programs. This is useful for comparing with trace output (see `trace161`) to ensure you're mapping the right pages or to locate the source of exceptions. You can also disassemble within gdb by using the `disassemble` command or `x/i`.

Usage:

To disassemble a program:

```
cs161-objdump --disassemble --source program
```

If you compiled the program with debug information, you can do the following to mix the source code in with the assembly:

```
cs161-objdump --disassemble --source program
```

Make sure you are actually disassembling the same copy of the program that you are running, or you will get very confused.

2.3.3 `cs161-addr2line`

Description:

`cs161-addr2line` converts an address in hex to a line number, so you can figure out where your program crashed and burned. When the kernel crashes, it prints the offending EPC. Using `cs161-addr2line` on this EPC gives the line number of corresponding line of code.

Usage:

```
cs161-addr2line hexaddr
```

2.3.4 `trace161`

Description:

`trace161` is useful for tracing through the execution of both user level programs and the kernel. Unfortunately, we can't debug user level programs, but we can trace them. Use `ltrace_on()/ltrace_off()` to turn tracing on/off under software control (see below).

Usage:

Using `trace161` instead of `sys161`, you can have it report various execution details, right down to the level of individual instructions. `trace161` has various useful options (you can list these by just running `trace161`). The most commonly used ones are:

- `trace161 -tu kernel` - reports every instruction executed in user mode. Useful when compared with the disassembly of the program, to ensure that you are mapping the right pages into memory.
- `trace161 -tk kernel` - reports every instruction executed in kernel (NOTE: in general, produces too much output to be useful).

2.3.5 `ltrace_debug()`

Description:

`ltrace_debug` is useful for printing simple indications that a certain piece of code has been reached, like one might use `kprintf`, except that it is less invasive than `kprintf`. Think of it as setting the value of a readout on the system's front panel. (In real life, since computers don't have front panels with blinking lights any

more, people often use the speaker or the top left corner of the screen for this purpose.) Although we don't recommend `printf` debugging, sometimes it can be useful (e.g. examining every `vm_fault` breakpoint may be inefficient). In those cases, using `ltrace_debug` is less likely to cause the bug to disappear (e.g. race conditions can be difficult to find using `kprintf` since a `kprintf` call can change the scheduler behavior. `ltrace_debug` is less likely to cause this).

Usage:

`ltrace_debug(CODE)` causes `sys161/trace161` to print a message with `CODE`, where `CODE` is a `textttu_int32_t`.

2.3.6 `ltrace_on()/ltrace_off()`

Description:

Turn tracing on/off under software control using `ltrace_on()/ltrace_off()`.

Usage:

`ltrace_on()` turns on the `trace161` tracing flag `CODE`.

`ltrace_off()` turns off the `trace161` tracing flag `CODE`.

2.3.7 `kmalloc` heap integrity mode

Description: `kmalloc` has two pound defines, `SLOW` and `SLOWER`, that turn on kernel heap integrity checking (these are off by default, as they make the system quite slow). These will catch various kinds of `malloc` bugs and thus can be useful in debugging. (When you write your VM system, you might want to use a similar `SLOW` mechanism to introduce some consistency checks of your own.)

2.3.8 profiling

Description:

Profiling is useful for performance analysis and the call graph can be useful for debugging. Check out `gprof`.