Using the GNU Debugger

6.828 Fall 2018

September 12, 2018

Homework solution

Homework solution

From bootasm.S:

```
# Set up the stack pointer and call into C.
movl    $start, %esp
call    bootmain
```

Homework solution

From bootasm.S:

```
# Set up the stack pointer and call into C.
movl    $start, %esp
call    bootmain
```

Later, in `bootmain()`:

```
// Call the entry point from the ELF header.
// Does not return!
entry = (void(*)(void))(elf->entry);
entry();
```

What's on the stack?

What's on the stack?

- `call bootmain` pushes a return address

What's on the stack?

- `call bootmain` pushes a return address
- The prologue in `bootmain()` makes a stack frame

What's on the stack?

- `call bootmain` pushes a return address
- The prologue in `bootmain()` makes a stack frame

```
push   %ebp
mov    %esp,%ebp
push   %edi
push   %esi
push   %ebx
sub    $0x1c,%esp
```

What's on the stack?

- `call bootmain` pushes a return address
- The prologue in `bootmain()` makes a stack frame

```
push   %ebp
mov    %esp,%ebp
push   %edi
push   %esi
push   %ebx
sub    $0x1c,%esp
```

- The call to `entry()` pushes a return address

The stack when we get to 0x0010000c

0x7c00:	0x8ec031fa	not the stack!
0x7bfc:	0x00007c4d	bootmain() return address
0x7bf8:	0x00000000	old ebp
0x7bf4:	0x00000000	old edi
0x7bf0:	0x00000000	old esi
0x7bec:	0x00000000	old ebx
0x7be8:	0x00000000	
0x7be4:	0x00000000	
0x7be0:	0x00000000	
0x7bdc:	0x00000000	local vars (sub $0x1c,%esp)
0x7bd8:	0x00000000	
0x7bd4:	0x00000000	
0x7bd0:	0x00000000	
0x7bcc:	0x00007db7	entry() return address

GDB in 6.828

We provide a file called `.gdbinit` which automatically sets up GDB for use with QEMU.

- Must run GDB from the lab or xv6 directory
- Edit `~/.gdbinit` to allow other gdbinits

GDB in 6.828

We provide a file called `.gdbinit` which automatically sets up GDB for use with QEMU.

- Must run GDB from the lab or xv6 directory
- Edit `~/.gdbinit` to allow other gdbinits

Use `make` to start QEMU with or without GDB.

- With GDB: run `make qemu[-nox]-gdb`, then start GDB in a second shell
- Use `make qemu[-nox]` when you don't need GDB

GDB commands

Run `help <command-name>` if you're not sure how to use a command.

GDB commands

Run `help <command-name>` if you're not sure how to use a command.

All commands may be abbreviated if unambiguous:

`c` = `co` = `cont` = `continue`

Some additional abbreviations are defined, e.g.

`s` = `step` and `si` = `stepi`

Stepping

`step` runs one line of code at a time. When there is a function call, it steps *into* the called function.

Stepping

`step` runs one line of code at a time. When there is a function call, it steps *into* the called function.

`next` does the same thing, except that it steps *over* function calls.

Stepping

`step` runs one line of code at a time. When there is a function call, it steps *into* the called function.

`next` does the same thing, except that it steps *over* function calls.

`stepi` and `nexti` do the same thing for assembly instructions rather than lines of code.

Stepping

`step` runs one line of code at a time. When there is a function call, it steps *into* the called function.

`next` does the same thing, except that it steps *over* function calls.

`stepi` and `nexti` do the same thing for assembly instructions rather than lines of code.

All take a numerical argument to specify repetition. Pressing the enter key repeats the previous command.

Running

`continue` runs code until a breakpoint is encountered or you interrupt it with Control-C.

Running

`continue` runs code until a breakpoint is encountered or you interrupt it with Control-C.

`finish` runs code until the current function returns.

Running

`continue` runs code until a breakpoint is encountered or you interrupt it with Control-C.

`finish` runs code until the current function returns.

`advance <location>` runs code until the instruction pointer gets to the specified location.

Breakpoints

`break <location>` sets a breakpoint at the specified location.

Breakpoints

`break <location>` sets a breakpoint at the specified location.

Locations can be memory addresses (“`*0x7c00`”) or names (“`mon_backtrace`”, “`monitor.c:71`”).

Breakpoints

`break <location>` sets a breakpoint at the specified location.

Locations can be memory addresses ("`*0x7c00`") or names ("`mon_backtrace`", "`monitor.c:71`").

Modify breakpoints using `delete`, `disable`, `enable`.

Conditional breakpoints

`break <location> if <condition>` sets a breakpoint at the specified location, but only breaks if the condition is satisfied.

Conditional breakpoints

`break <location> if <condition>` sets a breakpoint at the specified location, but only breaks if the condition is satisfied.

`cond <number> <condition>` adds a condition on an existing breakpoint.

Watchpoints

Like breakpoints, but with more complicated conditions.

Watchpoints

Like breakpoints, but with more complicated conditions.

`watch <expression>` will stop execution whenever the expression's value changes.

Watchpoints

Like breakpoints, but with more complicated conditions.

`watch <expression>` will stop execution whenever the expression's value changes.

`watch -l <address>` will stop execution whenever the contents of the specified memory address change.

Watchpoints

Like breakpoints, but with more complicated conditions.

`watch <expression>` will stop execution whenever the expression's value changes.

`watch -l <address>` will stop execution whenever the contents of the specified memory address change.

What's the difference between `wa var` and `wa -l &var`?

Watchpoints

Like breakpoints, but with more complicated conditions.

`watch <expression>` will stop execution whenever the expression's value changes.

`watch -l <address>` will stop execution whenever the contents of the specified memory address change.

What's the difference between `wa var` and `wa -l &var`?

`rwatch [-l] <expression>` will stop execution whenever the value of the expression is read.

Examining

`x` prints the raw contents of memory in whatever format you specify (`x/x` for hexadecimal, `x/i` for assembly, etc).

Examining

`x` prints the raw contents of memory in whatever format you specify (`x/x` for hexadecimal, `x/i` for assembly, etc).

`print` evaluates a C expression and prints the result as its proper type. It is often more useful than `x`.

Examining

`x` prints the raw contents of memory in whatever format you specify (`x/x` for hexadecimal, `x/i` for assembly, etc).

`print` evaluates a C expression and prints the result as its proper type. It is often more useful than `x`.

The output from `p *((struct elfhdr *) 0x10000)` is much nicer than the output from `x/13x 0x10000`.

More examining

`info registers` prints the value of every register.

More examining

`info registers` prints the value of every register.

`info frame` prints the current stack frame.

More examining

`info registers` prints the value of every register.

`info frame` prints the current stack frame.

`list <location>` prints the source code of the function at the specified location.

More examining

`info registers` prints the value of every register.

`info frame` prints the current stack frame.

`list <location>` prints the source code of the function at the specified location.

`backtrace` might be useful as you work on lab 1!

Layouts

GDB has a text user interface that shows useful information like code listing, disassembly, and register contents in a curses UI.

`layout <name>` switches to the given layout.

Other tricks

You can use the `set` command to change the value of a variable during execution.

Other tricks

You can use the `set` command to change the value of a variable during execution.

You have to switch symbol files to get function and variable names for environments other than the kernel. For example, when debugging JOS:

```
symbol-file obj/user/<name>
symbol-file obj/kern/kernel
```

Summary

Read the fine manual! Use the `help` command.

Summary

Read the fine manual! Use the `help` command.

GDB is tremendously powerful and we've only scratched the surface today.

Summary

Read the fine manual! Use the `help` command.

GDB is tremendously powerful and we've only scratched the surface today.

It is well worth your time to spend an hour learning more about how to use it.