Using the GNU Debugger

6.828 Fall 2018

September 12, 2018
Homework solution

From bootasm.S:

```asm
movl $start, %esp
call bootmain
```

Later, in `bootmain()`:

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

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.
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
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7c00</td>
<td>0x8ec031fa</td>
<td>not the stack!</td>
</tr>
<tr>
<td>0x7bfc</td>
<td>0x00007c4d</td>
<td>bootmain() return address</td>
</tr>
<tr>
<td>0x7bf8</td>
<td>0x00000000</td>
<td>old ebp</td>
</tr>
<tr>
<td>0x7bf4</td>
<td>0x00000000</td>
<td>old edi</td>
</tr>
<tr>
<td>0x7bf0</td>
<td>0x00000000</td>
<td>old esi</td>
</tr>
<tr>
<td>0x7bec</td>
<td>0x00000000</td>
<td>old ebx</td>
</tr>
<tr>
<td>0x7be8</td>
<td>0x00000000</td>
<td>local vars (sub $0x1c,%esp)</td>
</tr>
<tr>
<td>0x7be4</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x7be0</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x7bd8</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x7bd4</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x7bd0</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x7bcc</td>
<td>0x000007db7</td>
<td>entry() return address</td>
</tr>
</tbody>
</table>
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
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 gdb initiators

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.

\[ s = step \quad \text{and} \quad si = stepi \]
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.

\texttt{watch \langle expression\rangle} will stop execution whenever the expression’s value changes.

\texttt{watch -l \langle address\rangle} will stop execution whenever the contents of the specified memory address change.

What’s the difference between \texttt{wa \langle var\rangle} and \texttt{wa -l &\langle var\rangle}?

\texttt{rwatch [-l] \langle expression\rangle} will stop execution whenever the value of the expression is read.
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.
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

\texttt{x} prints the raw contents of memory in whatever format you specify (\texttt{x/x} for hexadecimal, \texttt{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

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

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

The output from \texttt{p *((struct elfhdr *) 0x10000)} is much nicer than the output from \texttt{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.