Debugging

GDB, the GNU Debugger, is a program that makes it easy to inspect the execution of other programs. We will be using it extensively in this class.

Opening GDB

To invoke GDB on an executable named `a.out`, run (in the shell)

```
gdb a.out
```

Alternatively, you can start GDB without specifying an executable, then select an executable using `gdb`'s `file` command:

```
gdb
(gdb) file a.out
```

Running Programs in GDB

Once a program is loaded in GDB, you can run it with the `run` (or `r`) command:

```
(gdb) run
```

If you want to pass command line arguments to your program, you can pass them to `run`. For example, to run the currently-loaded program with arguments `a`, `b`, and `c`, use

```
(gdb) run a b c
```

If you want to pass input to your program on `stdin`, you can use process substitution. For example, to run the currently-loaded program with input `ABCDEF` on `stdin`, run

```
(gdb) run < <(printf 'ABCDEF')
```

You might also want to start a program, then pause its execution just before the first instruction executes. You can accomplish this as follows:

```
(gdb) starti
```

Note that for some programs (dynamically-linked programs) the location of this first instruction might be surprising!

Breakpoints

To pause a program's execution at a particular program point, make a breakpoint!

For example, to pause `a.out`'s execution at the beginning of `main`[1], run

```
(gdb) break main
```

Alternatively, if the address of main is `0x800000`, you can also use:

```
(gdb) break *0x800000
```

notice the `*` symbol used to indicate an address.

Then, when you execute the `run` command, you'll be dropped back into the GDB prompt, and can further inspect the program's state.

You might also want to pause only when a particular condition is true. For example, to set a breakpoint at the beginning of `main` that activates only when the `edi` register is `1`, run

```
(gdb) break main if $edi == 1
```

To list all currently-set breakpoints, run

```
(gdb) info breakpoints
```

To remove a breakpoint, use `delete` or `d`. For example, to delete the first breakpoint created during this debugging session, use

```
(gdb) delete 1
```

Alternatively, you can delete all the breakpoints with a plain

[1]This will actually set a breakpoint just after `main`'s function prologue, but close enough :)

```
(gdb) delete
```

Resuming Program Execution

To resume program execution after stopping at a breakpoint, use the `continue` (or `c`) command. If you keep hitting the same breakpoint, and want to skip it 10 times in a row, run

```
(gdb) continue 10
```

Stepping

Once you've hit a breakpoint, you can execute a single instruction using the `stepi` (or `si`) command. For example, to run only the first instruction in `main` (after its prologue), you might do the following:

```
(gdb) break main
(gdb) run
(gdb) stepi
```

You may find that `stepi` is too fine-grained, particularly when debugging functions that call many other functions, because `stepi` executes everything one instruction at a time. In that scenario, consider using the `nexti` command, which is just like `stepi`, but if the current instruction is a `call`, it automatically continues execution until the called function returns.

Disassembling

To disassemble instructions starting from `rip` in `gdb`, use the `disas` command.

```
(gdb) b main
Breakpoint 1 at 0x113d
(gdb) run
Starting program: /home/bkallus/a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 1, 0x000055555555513d in main ()
(gdb) disas
Dump of assembler code for function main:
   0x0000555555555139 <+0>: push   rbp
   0x000055555555513a <+1>: mov    rbp,rsp
=> 0x000055555555513d <+4>: lea    rax,[rip+0xec0]        # 0x555555556004
   0x0000555555555144 <+11>:    mov    rdi,rax
   0x0000555555555147 <+14>:    mov    eax,0x0
   0x000055555555514c <+19>:    call   0x555555555030 <printf@plt>
   0x0000555555555151 <+24>:    mov    eax,0x0
   0x0000555555555156 <+29>:    pop    rbp
   0x0000555555555157 <+30>:    ret
End of assembler dump.
```

To disassemble a function that is not currently executing, pass its name as an argument to `disas`. For example, to disassemble `main`, run `disas main`.

Inspecting Registers

You can examine the current state of the registers as follows:

```
(gdb) info registers
rax            0x7ffff7ffe2d8      140737354130136
rbx            0x0                 0
rcx            0x7ffff7fc5000      140737353895936
rdx            0x0                 0
rsi            0x0                 0
rdi            0x7fffffffd5e0      140737488344544
rbp            0x0                 0x0
rsp            0x7fffffffd5d0      0x7fffffffd5d0
...
```

Inspecting Memory

The `x` command is used to examine memory. The command has the following format:

```
x/[Amount to Read][Format of Read][Unit Size] [Address]
```

For example, to show **3** he**x**adecimal **b**ytes from `0x7ffff7ffe2d8`, run

```
(gdb) x/3xb 0x7ffff7ffe2d8
0x7ffff7ffe2d8: 0xe8    0xe6    0xff
```

Here are some format specifiers you may find useful:

- `x` (hexademical)
- `i` (instruction)
- `s` (string)

Here are some size units you may find useful:

- `b` (1 byte)
- `h` (2 bytes)
- `w` (4 bytes)
- `g` (8 bytes)

Setting Registers

To set a register to a new value, use the `set` command. For example, to set `eax` to 5, run

```
(gdb) set $eax = 5
```

Getting Help

If you're unsure about how to use a command, use the `help` command. For example, to see more information about how the `nexti` command works, you might try

```
(gdb) help nexti
```

Quirks

- Hitting enter on an empty prompt will re-run the previous command.
- Sometimes a `*` is needed before address literals, even when there is no dereference occurring. This is very unintuitive.