Project 1 Walkthrough

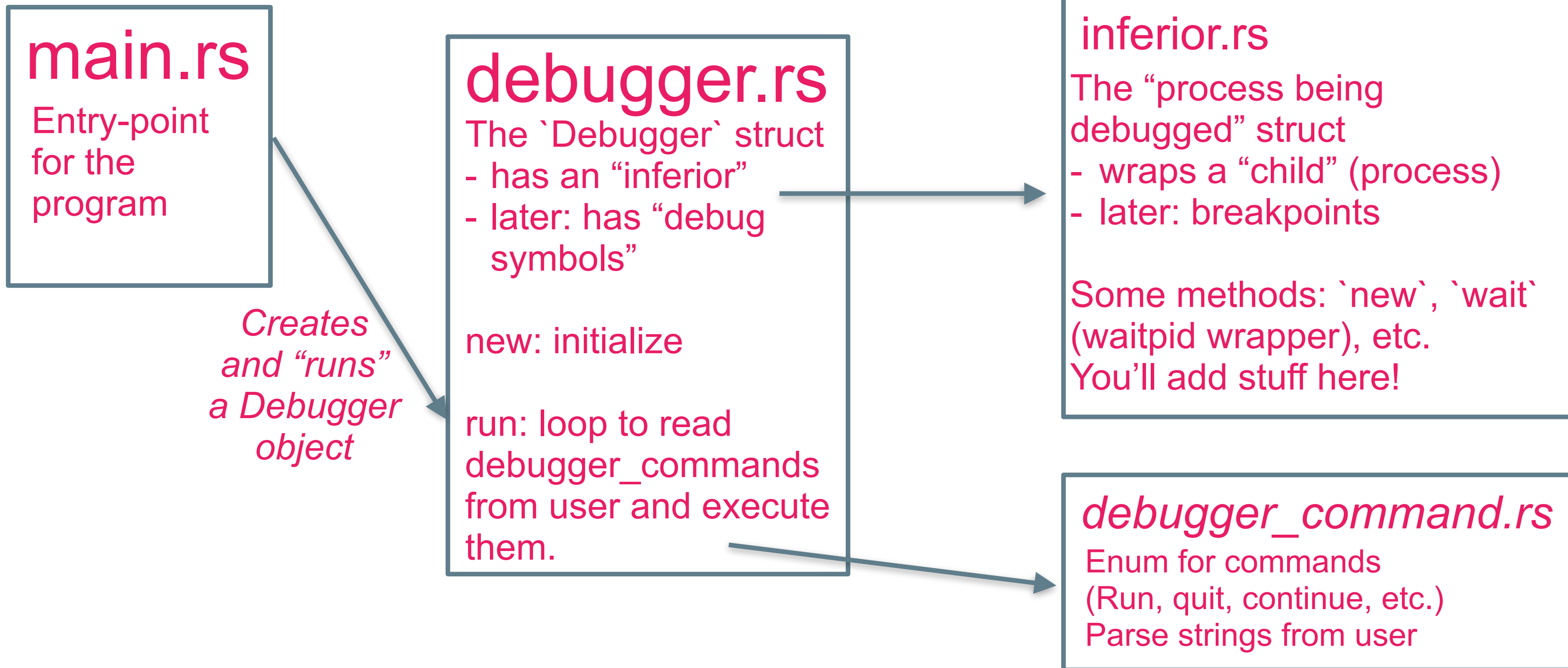
CS110L
February 7, 2022

Getting set up

- Please accept your repo invitations now (before they expire)!!!
- Working together:
  - You can work individually or in groups of 2-3
  - Working in groups: you can submit together or separately
  - See instructions on project 1 handout and collaboration tools tips.
  - Submit on GitHub.
- Getting a Linux setup:
  - Unfortunately, the process interface we use is specific to Linux.
  - Myth machines seem to work. More options for getting a Linux setup on your own machine are on the website.
  - This is the last assignment where you'll need Linux!

Milestone 0:
Check Out the Starter Code

Starter code



Milestone 1:
Start the “Inferior” as a Child Process

Milestone 1: start the “inferior” as a child process

```rust
fn main() {
    // parse args from user -> get `target` to debug
    let mut debugger = Debugger::new(target);
    debugger.run();
}
```

Milestone 1: start the “inferior” as a child process

```
fn main() {
    // parse args from user -> get `target` to debug
    let mut debugger = Debugger::new(target);
    debugger.run();
}
```

LOOP:

- *Read command from user (e.g., “run” or “quit”)*
- *Execute the command*

```
🍌 cargo run samples/sleepy_print
    Finished dev [unoptimized + de
     Running `target/debug/deet sa
(deet) r 3
0
1
2
Child exited (status 0)
```

```
loop {
    match self.get_next_command() {
        DebuggerCommand::Run(args) => {
            // do something
        }
        DebuggerCommand::Quit => {
            // do something else
        }
    }
}
```

Milestone 1: start the “inferior” as a child process

```
DebuggerCommand::Run(args) => {
    // TODO (milestone 1): implement Inferior::new
    if let Some(inferior) = Inferior::new(&self.target, &args) {
        self.inferior = Some(inferior);
        // TODO (milestone 1): make the inferior run
     } else {
        println!("Error starting subprocess");
}
```

- What this code does now: when a user enters “run”, calls a dummy function that returns an Option<Inferior>. If that Option is Some, stores the encapsulated `inferior` in the Debugger struct. Else, prints an error.

Milestone 1: start the “inferior” as a child process

- Inferior::new should start a new inferior process:
  - Construct a **Command::new** with the given **target** and **args**
  - Set the **pre_exec function** to be **child_traceme**
    - This calls the **ptrace** system call with **PTRACE_TRACEME**
    - Enables debugging on the child process from the parent process
  - **Wait** for the child process to successfully start and stop
    - Any process started with PTRACE_TRACEME will “stop” with a SIGTRAP as soon as it starts
  - **Verify** that the child process has, in fact, stopped with SIGTRAP
- If Inferior::new returns successfully, **continue** the process and wait for it to stop or exit.

Milestone 3:
Print a Backtrace

Skipping overview of Milestone 2, which involves implementing `cont` and (I recommend) some decomposition of your code from part 1.

Milestone 3: implementing a backtrace

- Background: compiling a program
  - When a program is compiled, its executable instructions end up in the **text segment** of a process. (In other words, instructions are stored in memory, and they have memory addresses — just like normal variables.)
  - Instruction addresses are stored in the **%rip (instruction pointer) register** as they are executed.
- When a program is compiled for **debug mode**, extra **debugging symbols** are stored within the executable.
  - These help map **memory addresses** to functions, line numbers, file names, variables, and more.

Milestone 3: implementing a backtrace

- One more piece of starter code: **dwarf_data.rs**
- Wrapper around the DWARF debugging format and the gimli library
- Implements a **struct DwarfData**:
    - Contains debug symbols (mappings) generated from a target file.
    - Has some helpful functions, e.g.:
        - **get_line_from_addr(&self, curr_addr: u64):** given the memory address of an instruction, returns the file and line number it corresponds to in source code.
        - **get_function_from_addr(&self, curr_addr: u64):** given the memory address of an instruction, returns the name of the function it's in.

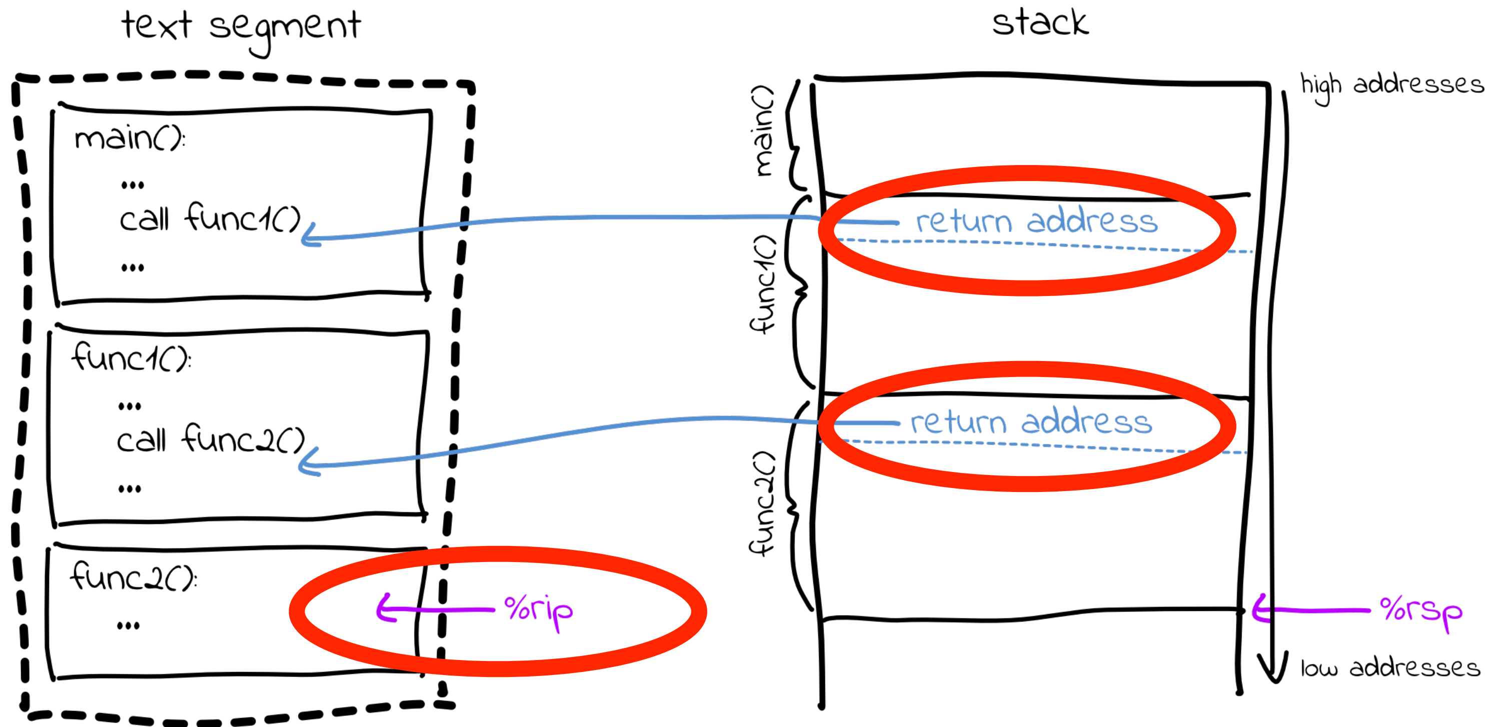
Milestone 3: implementing a backtrace

- Putting it all together…
- You can use **ptrace::getregs** to get all registers in the “traced” process (the process you’re debugging)
- You can get the value in the instruction pointer (%**rip**)
- You can pass that value into DwarfData’s mappings, which will give you source code level information to print!

```
(deet) r
Calling func2
About to segfault... a=2
Child stopped (signal SIGSEGV)
(deet) back
func2 (/deet/samples/segfault.c:5)
(deet)
```

Milestone 3: implementing a backtrace

- To turn this into a full backtrace, we need to “travel up the stack”:



Milestone 3: implementing a backtrace

- func2 was called by func1, so return address at top of func2's stack frame will take us to func1
- func1 was called by main, so return address at top of func1's stack frame will take us to main
- Once we hit main, break.

```
(deet) r
Calling func2
About to segfault... a=2
Child stopped (signal SIGSEGV)
Stopped at /deet/samples/segfault.c:5
(deet) back
func2 (/deet/samples/segfault.c:5)
func1 (/deet/samples/segfault.c:12)
main (/deet/samples/segfault.c:15)
(deet)
```

Milestones 5, 6, and 7: Setting Breakpoints

Skipping overview of Milestone 4, which involves applying the same concepts as milestone 3.

Milestone 5: Breakpoints on Memory Addresses

- Add a **break** command.
  - For now, this should take one argument: a memory address.
    - (e.g., **break *0x123456**)
- If breakpoints are set before the process is running, store them in the Debugger, and pass them to **Inferior::new** when the user **runs** the inferior.
- In Inferior::new, install the breakpoints.
  - *How?*

```
(deet) break *0x400b6d
Set breakpoint 0 at 0x400b6d
(deet) r
Calling func2
Child stopped (signal SIGTRAP)
Stopped at /deet/samples/segfault.c:3
```

Background: Installing Breakpoints

- The executable instructions of a program are stored in a read-only segment of a process' virtual memory called the **text segment**.
  - Instructions are just *values written to memory*.
  - Every instruction has a *memory address* and a *size*.
- **Machine-encoded instructions** are represented as one or more **bytes**
  - Ex: push = 0x55, mov = 0x89, ret = 0xc3
  - These "codes" are what's actually stored in the text segment of a process — what the CPU reads and interprets as instructions.
- **%rip,** the instruction pointer, stores the memory address of the "next instruction to execute".

Milestone 5: Breakpoints on Memory Addresses

How breakpoints work in gdb:

- Manually **replace the instruction** you want to break at with the “**interrupt**” instruction (0xcc).
- E.g.: if you want to set a breakpoint on the instruction stored at 0x123456, use **ptrace** to **write** the byte **0xcc** to 0x123456 in the child process.
- When the child process gets to the interrupt instruction (%rip => 0x123456), it will temporarily halt. Parent can examine using waitpid.

Milestone 5: Breakpoints on Memory Addresses

Summary:

- There should be a **break** enum variant in **DebuggerCommand**.
- When triggered in **run** loop: parse a valid memory address provided for the breakpoint.
  - If the inferior has already started running, **install the breakpoint**.
  - Otherwise, store it in the Debugger to be installed later.
- When inferior is created, install all stored breakpoints (if any).
- To install a breakpoint at location **X**: write_byte 0xcc to location **X**
  - Optionally, print a confirmation message
- *Note: for testing, add a call to `debug_data.print()` in Debugger::new to print out debug symbols (e.g., memory addresses <-> line numbers).*

Milestone 6: Continuing from a Breakpoint

- Problem: to set a breakpoint, we **overwrote the first byte of a valid instruction in the program**. If we continue, this could cause… issues.
- To continue from a breakpoint:
  - Replace 0xcc with the **original instruction's value** (note: you'll probably need to have this stored when breakpoints are set)
  - **Rewind** the instruction pointer in %rip to before the breakpoint (set %rip to %rip - 1)
  - Execute that instruction: tell **ptrace** to continue by just one instruction
  - Restore the breakpoint: replace the instruction with **0xcc** again
  - Resume normal execution

Milestone 6: Continuing from a Breakpoint

1. Breakpoint hit,
   instruction pointer incremented

	Address	Contents
	0x123453	mov
	0x123454	%esp,
	0x123455	%ebp
	0x123456	**INTERRUPT**
%rip →	0x123457	$4
	0x123458	%esp
	0x123459	push
	0x123460	%edi

2. Restore original first byte of instruction

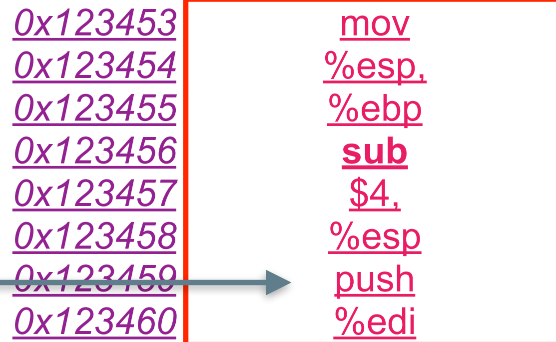
	Address	Contents
	0x123453	mov
	0x123454	%esp,
	0x123455	%ebp
	0x123456	**sub**
%rip →	0x123457	$4,
	0x123458	%esp
	0x123459	push
	0x123460	%edi

3. “Roll back” instruction pointer
   to beginning of true instruction

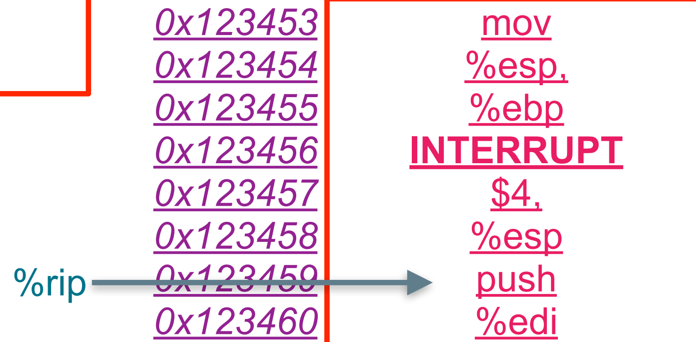
	Address	Contents
	0x123453	mov
	0x123454	%esp,
	0x123455	%ebp
%rip →	0x123456	**sub**
	0x123457	$4,
	0x123458	%esp
	0x123459	push
	0x123460	%edi

Milestone 6: Continuing from a Breakpoint

4. Execute a single instruction



5. Restore the breakpoint



6. Continue normal execution

Milestone 7: Setting Breakpoints on Symbols

- Apply what you know about DwarfData from milestone 3 to allow users to **break on functions or line numbers**!
- I.e.: translate line number or function name to an address, then call your code from milestones 5/6.

```
🍌 cargo run samples/segfault
   Compiling deet v0.1.0 (/deet)
    Finished dev [unoptimized + debuginfo] target(s) in 26.91s
     Running `target/debug/deet samples/segfault`
(deet) break 15
Set breakpoint 0 at 0x400bf1
(deet) break func1
Set breakpoint 1 at 0x400bad
(deet) break func2
Set breakpoint 2 at 0x400b71
(deet) r
Child stopped (signal SIGTRAP)
Stopped at /deet/samples/segfault.c:15
(deet) c
Child stopped (signal SIGTRAP)
Stopped at /deet/samples/segfault.c:9
```

Aside: a few tips

A few tips

- Error handling:
  - The `?` operator is really useful for propagating errors. (Notes here.)
  - The `Result::ok()` method is useful for converting **Results** to **Options**. (Notes in the week 3 exercises here.)
  - Possible example: `let child = command.spawn().ok()?;`
- Read documentation
  - We're using the `nix` library — interface for libc functions like **waitpid** and **ptrace**. If you're using these, make sure you know what parameters these take in and what types they return.

A few tips

- `pub` structs, members of structs, enums, functions, etc.
  - Pub = “this interface is accessible from outside of this module”
  - Default = “no one outside of this module can invoke this interface”
  - E.g., if you want a function in `inferior.rs` to be callable from debugger.rs, mark it as `pub`. If you make a helper function in inferior.rs that’s only used internally, don’t make it `pub`.
- Use keyword:
  - You may have to import items from other modules/crats to use them
  - Ex: to use Command, you’ll need **use std::process::Command**
  - These are noted on the handout. If you get a “cannot find X in this scope” error, make sure you’ve imported what you need.

A few tips

- **Option::as_ref** and **Option::as_mut**
- Talked about **as_ref** in [**lecture 6**]
- Both as_ref and **as_mut** are in the [**lecture 7**] notes
- **as_ref** converts &Option<T> -> Option<&T>
- as_mut is the same as as_ref, but with *mutable* references.
- Docs [here]
- Example usage, in Debugger::run:

```
DebuggerCommand::Run(args) => {
   if let Some(inferior) = Inferior::new(&self.target, &args) {
      self.inferior = Some(inferior);
      let inf = self.inferior.as_mut().unwrap();
/// ...
```

Note: some compiler magic happening here.
`self.inferior.as_mut().unwrap();`
in this code snippet is expanded into
`(&self.inferior).as_mut().unwrap()`

A few tips

- Get comfortable with `enums` and `match` expressions.
- Example:

```rust
pub enum Status {
    /// Indicates inferior stopped.
    /// Contains the signal that stopped the
    /// process, as well as the current
    /// instruction pointer it is stopped at.
    Stopped(signal::Signal, u64),
    /// Indicates inferior exited normally.
    /// Contains the exit status code.
    Exited(i32),
    /// Indicates the inferior exited due to a
    /// signal. Contains the signal that
    /// killed the process.
    Signaled(signal::Signal),
}
```

A few tips

- Match expression example #1:

```
match status {
    Status::Stopped(signal::SIGTRAP, _) => {}
    _ => return None,
}
```

If `status` is of type Status::Stopped *and* the encapsulated `signal` is of type `signal::SIGTRAP`, do nothing

In all other situations (default option), return None.

A few tips

- Match expression example #2:

If the `status` is of type Status::Stoppe
- Capture the `sig` (Signal) encapsulated in the Status

```
match status {
    Status::Stopped(sig, _) => {
        // do something, possibly involving `sig`
    }
    Status::Exited(stat) => {
        // do something, possibly involving `status`
    }
    // etc.
}
```

If the `status` is of type Status::Exited
- Capture the exit status code encapsulated

That’s it!

Feel free to implement any additional functionality that interests you :)

E.g.: “next” command, print source code, print variables…