User Documentation/Manual: Minimal Debugger

Version: 1.0
Author: Jaskaran Singh
Date: December 17, 2025.

1. Overview

Minidbg is a lightweight, custom UNIX-style debugger integrated directly into the myshell command-line interpreter. This project fulfills the requirements for Lab 2, providing low-level process control and inspection capabilities.
Unlike standard shells that simply execute programs, Minidbg allows the user to act as a parent process that traces a target child process. It supports setting breakpoints, single-stepping, inspecting CPU registers, and viewing raw memory, all without relying on external tools like GDB.

2. System Requirements

- **Operating System:** macOS (Intel or Apple Silicon) / UNIX-based OS.
    - *Note:* This implementation uses specific **Mach Kernel APIs** (`mach_vm_protect`, `task_for_pid`) required for debugging on macOS.
- **Compiler:** Clang or GCC.
- **Build Tool:** GNU Make.
- **Privileges: Root access (sudo) is strictly required.**
    - macOS security policies prevent standard users from controlling other processes or inspecting their memory.

3. Installation and Compilation

The project is distributed as C source code and includes a `Makefile` for automated building.

Step 1: Clean and Compile Open your terminal, navigate to the project directory, and run:

```
make clean
make
```

- **Success:** This generates two executables:
    - `shell`: The main shell and debugger interface.

- `target`: A test program designed to demonstrate debugging features.

Step 2: Launching with Privileges Because the debugger requires access to kernel structures, you must start the shell using `sudo`:

```
sudo ./shell
```

You will see the prompt:


```
sudo ./shell
```


4. Debugger Feature Guide

The debugger is accessed via the `debug` command from within the custom shell.

4.1 Starting a Debug Session

To load a program under the debugger's control: **Syntax:** `debug <executable_path>` **Example:**

```
myshell> debug ./target
Debugger started. Type 'break <addr>', 'continue', or 'quit'.
```

- **Result:** The shell launches the target process, pauses it immediately, and switches the prompt to `minidbg>`, indicating you are now in **Debugger Mode**.

4.2 Controlling Execution

Once in Debugger Mode, you can control the flow of the child process.

- **`continue`**
  - Resumes execution of the target program. The program will run until it hits a breakpoint, encounters a signal, or exits.
  - **Usage:** `minidbg> continue`
- **`quit`**
  - Terminates the debugger session, kills the child process, and returns you to the standard `myshell>` prompt.
  - **Usage:** `minidbg> quit`

4.3 Breakpoints

You can pause execution at specific memory addresses to inspect the program state.

- **`break <address>`**
  - Sets a software breakpoint at the specified hexadecimal address.
  - **Mechanism:** This command transparently writes the `INT 3` (opcode `0xCC`) trap instruction into the target's memory.
  - **Usage:** `minidbg> break 0x100003f60`

4.4 State Inspection

When the target is paused (e.g., after hitting a breakpoint), you can inspect its internal state.

- **`regs`**
  - Displays the current values of the CPU registers.
  - **Registers Shown:** Instruction Pointer (`RIP`), Stack Pointer (`RSP`), Base Pointer (`RBP`), and Accumulator (`RAX`).
  - **Usage:** `minidbg> regs`
- **`peek <address>` (Extension)**
  - Reads and displays 4 bytes of raw data from the specified memory address.
  - **Usage:** `minidbg> peek 0x100003f60`

5. Walkthrough: Debugging a Target

This tutorial demonstrates debugging the included `target.c` program. Note that due to **ASLR (Address Space Layout Randomization)**, memory addresses will change every time you run the program.

Step 1: Start the Session

```
myshell> debug ./target
Debugger started. Type 'break <addr>', 'continue', or 'quit'.
```

Step 2: Determine the Address The `target` program is programmed to print the address of its function `foo`. Run it once to see this address.

```
minidbg> continue
Resuming execution...
Target started.
ADDRESS_OF_FOO: 0x10f28a470   <-- COPY THIS ADDRESS
Pausing for debugger... (raising SIGTRAP)
```

Step 3: Set the Breakpoint Use the address you just copied to set a trap.

```
minidbg> break 0x10f28a470
Breakpoint set at 0x10f28a470
```

Step 4: Trigger the Breakpoint Resume execution. The program will call `foo`, hitting your trap.

```
minidbg> continue
Resuming execution...
Hit breakpoint!
```

Step 5: Inspect Data Check the registers to confirm `RIP` matches your breakpoint, or peek at memory.

```
minidbg> regs
--- CPU Registers ---
RIP: 0x10f28a470
...
minidbg> peek 0x10f28a470
Data at 0x10f28a470: 0xffffffff
```

6. Troubleshooting / FAQ

Q: I get "Failed to get task for pid" or "Invalid argument" errors.
A: This usually happens if you did not run the shell with sudo. macOS requires root privileges to use task_for_pid and inspect other processes. Exit and restart with sudo ./shell.

Q: Why does the address of foo change every time?
A: This is due to ASLR (Address Space Layout Randomization), a security feature that randomizes where code is loaded in memory. You must always check the output of the current run to find the correct address for breakpoints.

Q: My breakpoint wasn't hit.
A: Ensure you set the breakpoint after the address was printed but before the function was called. In the target example, the program pauses itself specifically to give you time to set this breakpoint.

Q: Can I use standard shell commands while debugging?
A: No. When the prompt reads minidbg>, you are in a specialized input loop that only accepts debugger commands. Type quit to return to myshell> if you need to run ls or pwd.

7. Example Session

Below is a transcript of a typical debug session:

```
(base) ➜  Cornerstone-Project-col7001 git:(feature/debug) ✗ sudo ./shell
myshell>
myshell>
myshell> debug ./target
Starting debugger for ./target...
Debugger started. Type 'break <addr>', 'continue', or 'quit'.
minidbg>
minidbg> continue
Resuming execution...
Target started.
ADDRESS_OF_FOO: 0x106fa8470
Pausing for debugger... (raising SIGTRAP)
Hit breakpoint!
--- CPU Registers ---
RIP: 0x7ff819b2b82e
RSP: 0x7ff7b8f571f8
RBP: 0x7ff7b8f57220
RAX: 0x0
----------------------
minidbg>
minidbg> break 0x106fa8470
Breakpoint set at 0x106fa8470
minidbg>
minidbg>
minidbg> continue
Resuming execution...
Hit breakpoint!
--- CPU Registers ---
RIP: 0x106fa8471
RSP: 0x7ff7b8f57248
RBP: 0x7ff7b8f57260
RAX: 0x0
----------------------
minidbg>
minidbg>
minidbg> regs
--- CPU Registers ---
RIP: 0x106fa8471
RSP: 0x7ff7b8f57248
RBP: 0x7ff7b8f57260
RAX: 0x0
----------------------
minidbg>
minidbg> peek 0x106fa8470
Data at 0x106fa8470: 0xffffffff
minidbg>
minidbg>
minidbg> quit
myshell> exit
(base) ➜  Cornerstone-Project-col7001 git:(feature/debug) ✗
```

<---------- END OF REPORT ---------->