Mini-Debugger

A lightweight, educational debugger for Linux systems built using the ptrace system call. This project demonstrates fundamental debugging concepts including breakpoint management, single-stepping, register inspection, and process control.

Git Repo: https://github.com/aman-singh-12647/mini-debugger.git

Overview

Mini-Debugger is a command-line debugging tool that allows you to:

- Launch and control the execution of programs
- Set breakpoints at specific memory addresses
- Step through code instruction by instruction
- Inspect CPU registers
- Continue execution until breakpoints are hit

This debugger is designed for educational purposes to help understand how debuggers like GDB work under the hood. It uses Linux's ptrace API to control and inspect child processes.

System Requirements

Supported Systems

- Operating System: Linux (kernel 2.6+)
- Architecture: x86-64 (AMD64)
- Compiler: GCC 4.8+ or Clang 3.5+
- Build Tools: GNU Make

Dependencies

- Standard C library (glibc)
- Linux kernel headers
- POSIX-compliant system

Note: This debugger is specifically designed for Linux systems and will not work on macOS, Windows, or other Unix-like systems without significant modifications due to its reliance on Linux-specific ptrace functionality.

Installation

Step 1: Clone or Download the Repository

```
git clone https://github.com/aman-singh-12647/mini-debugger.git mini-debugger
cd mini-debugger
```

Step 2: Build the Debugger

```
make
```

This will compile all source files and create the debugger executable in the project root directory.

Step 3: Verify Installation

```
./debugger
```

You should see the usage message:

```
Usage: ./debugger <program>
```

Step 4: (Optional) Build Test Suite

```
make test
```

This compiles and runs the test suite to verify the debugger functionality.

Usage

Basic Syntax

```
./debugger <program_to_debug>
```

Where <program_to_debug> is the path to an executable you want to debug.

Basic Commands

Once the debugger starts, you'll see the dbg> prompt. Available commands:

Command	Syntax	Description
break	break <address>	Set a breakpoint at the specified hexadecimal address
cont	cont	Continue execution until a breakpoint is hit
step	step	Execute a single instruction (single-step)
regs	regs	Display all CPU registers and their current values
exit	exit	Quit the debugger

Example Session

1. Compile a Test Program

First, create a simple test program:

```
gcc -g -o test test.c
```

The -g flag includes debugging symbols, which helps identify memory addresses.

2. Find Function Addresses

Use objdump or nm to find function addresses:

```
objdump -d test | grep "<main>:"
```

Example output:

```
0000000000401136 <main>:
```

3. Start Debugging

```
./debugger ./test
```

4. Set a Breakpoint

```
dbg> break 401136
Breakpoint set at 0x401136
```

5. Continue Execution

```
dbg> cont
Hit breakpoint at 0x401136
```

6. Inspect Registers

```
dbg> regs
rax: 0x0000000000000000
rbx: 0x0000000000000000
rcx: 0x0000000000000000
...
```

7. Single Step

```
dbg> step
Stepped to next instruction
```

8. Exit

```
dbg> exit
```

Building and Testing

Build Commands

```
# Build the debugger
make

# Build and run tests
make test

# Clean build artifacts
make clean

# Clean only test artifacts
make clean-tests
```

Compiler Flags

The project uses the following GCC flags:

- `-Wall`: Enable all warnings
- `-g`: Include debugging information

Troubleshooting

Problem: "Permission denied" when running debugger

Cause: The debugger executable may not have execute permissions.

Solution:

```
chmod +x debugger
./debugger ./test
```

Problem: "ptrace: Operation not permitted"

Cause: Linux security restrictions may prevent ptrace from attaching to processes.

Solutions:

1. Check ptrace_scope setting:

```
cat /proc/sys/kernel/yama/ptrace_scope
```

If the value is `1` or higher, temporarily allow ptrace:

```
echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope
```

2. Run with appropriate permissions (not recommended for production):

```
sudo ./debugger ./test
```

Problem: Debugger hangs or doesn't respond

Cause: The debugged program may be waiting for input or in an infinite loop.

Solutions:

1. Press Ctrl+C to interrupt (may terminate both debugger and program)
2. Use a different terminal to kill the process:

```
ps aux | grep debugger
kill -9 <pid>
```

Problem: "make: command not found"

Cause: GNU Make is not installed.

Solution:

```
# Ubuntu/Debian
sudo apt-get install build-essential

# Fedora/RHEL
sudo dnf install make gcc

# Arch Linux
sudo pacman -S base-devel
```

Problem: Compilation errors about missing headers

Cause: Missing development headers or wrong architecture.

Solutions:

1. Install development tools:

```
# Ubuntu/Debian
sudo apt-get install build-essential linux-headers-$(uname -r)
```

2. Verify you're on x86-64:

```
uname -m
# Should output: x86_64
```