Introduction to Computing

Debugging using GDB

Introduction to GDB

What is GDB?

- GDB is a powerful debugger for programs written in C, C++, and other languages.

Why Use GDB?

- Debug runtime errors.
- Analyze program crashes.
- Inspect memory and variables.
- Step through code to understand execution flow.

Common Commands:

- Start debugging: `gdb <program_name>`
- Run program: `run` (or `r`)
- Exit GDB: `quit` (or `q`)

Setting Up GDB

Compile the Program for Debugging:

- Use the `-g` flag with `gcc` or `g++`:
- gcc -g program.c -o program

Starting GDB:

- Debug a program:
- gdb ./program
- Attach to a running process:
- gdb -p <pid>

Basic Setup Commands:

- Set breakpoint: `break <line_number>` or `break <function_name>`
- View source code: `list` or `l`

Running and Controlling Execution

Run the Program:

- Command: `run` or `r`
- Example: `run arg1 arg2`

Control Execution:

- Continue execution: `continue` or `c`
- Step into a function: `step` or `s`
- Step over a function: `next` or `n`
- Finish current function: `finish`

Interrupt Execution:

- Use `Ctrl+C` to pause the program at any point.

Inspecting Variables

Print Variable Values:

- Command: `print <variable>`
- print x

Inspect Memory:

- Command: `x <address>`
- x/4xw &array // View 4 words in hexadecimal

Watch Variables:

- Automatically stop when a variable changes:
- watch <variable>

Working with Breakpoints

Set Breakpoints:

- By line number: `break <line_number>`
- By function name: `break <function_name>`

Manage Breakpoints:

- List breakpoints: `info breakpoints`
- Disable breakpoint: `disable <breakpoint_number>`
- Enable breakpoint: `enable <breakpoint_number>`
- Delete breakpoint: `delete <breakpoint_number>`

Backtraces and Frames

Inspect Call Stack:

- Command: `backtrace` or `bt`
- Example Output:
- #0 main at program.c:12
- #1 helper_function at program.c:7

Navigate Frames:

- Switch to a specific frame: `frame <frame_number>`
- Go to the next frame: `up`
- Go to the previous frame: `down`

Advanced GDB Commands

Inspecting Assembly:

- Disassemble code: `disassemble`
- Example: `disassemble main`

Modify Variable Values:

- Command: `set variable <variable_name>=<value>`
- set variable x=42

Evaluate Expressions:

- Command: `print <expression>`
- print x + y

Log Output:

- Save GDB session to a file:
- set logging on

GDB TUI (Text User Interface)

Enable TUI Mode:

- Command: `layout src`
- Displays source code and execution in real-time.

Key Shortcuts:

- `Ctrl+L`: Refresh display.
- `Ctrl+X O`: Switch focus between code and command windows.

Exit TUI Mode:

- Command: `quit` or `Ctrl+X A`

Debugging Core Dumps

What is a Core Dump?

- A core dump captures a snapshot of the program's memory at the moment of a crash.

Steps to Enable Core Dumps:

- Enable core dumps in your shell:
- ulimit -c unlimited
- Run the program, allowing it to crash.

Debugging with GDB:

- Load the core dump:
- gdb <program> <core_file>
- Analyze the crash:
- `bt` (Backtrace): View the call stack at the time of the crash.
- Inspect variables to find the issue.

Conditional Breakpoints

What Are Conditional Breakpoints?

- A breakpoint that triggers only when a specified condition is met.

Syntax:

- break <line/function> if <condition>

Example:

- break 10 if x > 5

Commands for Managing Conditions:

- Add condition: `condition <breakpoint_number> <condition>`
- Remove condition: `condition <breakpoint_number>`

Debugging Multithreaded Programs

View Threads:

- Command: `info threads`
- Lists all active threads in the program.

Switch Between Threads:

- Command: `thread <thread_number>`
- Example: `thread 2`

Set Thread-Specific Breakpoints:

- Syntax:
- break <line/function> thread <thread_number>
- Example: `break 20 thread 3`

Examining Memory in Detail

Command: `x` (Examine Memory)

- Syntax:
- x/<count><format><size> <address>
- `<count>`: Number of items to view.
- `<format>`: Display format (`x` for hexadecimal, `d` for decimal, `s` for string, etc.).
- `<size>`: Data size (`b` for byte, `h` for halfword, `w` for word, `g` for giant word).

Example:

- x/4xw &array // View 4 words in hexadecimal

View Function Pointers:

- x/4i <address> // View instructions at address

GDB Scripting Basics

Automate GDB Commands:

- Use `.gdbinit` file for frequently used commands:
- break main
- run
- bt

Use GDB Scripts:

- Create a script file (e.g., `script.gdb`):
- break func
- run arg1 arg2
- bt

Run with GDB:

- gdb -x script.gdb ./program

Debugging Optimized Code

Challenge:

- Optimizations often reorder, inline, or eliminate code, making debugging harder.

Solution:

- Compile with debugging and optimization flags:
- gcc -g -O2 program.c -o program
- Use commands to map optimized code:
- `info line`: Locate line numbers.
- `disassemble`: View assembly instructions.

Tips:

- Use `step` and `next` cautiously, as lines may appear skipped.
- Inspect variable states with `info variables` or memory commands (`x`).

Debugging Libraries

Load Debug Symbols for Libraries:

- Install debugging symbols for libraries (e.g., `libc`):
- sudo apt install libc6-dbg

View Library-Specific Breakpoints:

- Command: `info sharedlibrary`
- Example: Set a breakpoint in `printf`:
- break printf

Step Through Library Code:

- Debug library internals with `step` and inspect library variables.

Practical Debugging Scenarios

Identify a Segmentation Fault:

- Debug this code:
- int *ptr = NULL;
- *ptr = 42; // Causes segmentation fault
- Use GDB to find the issue:
- Set breakpoints.
- Use `bt` to locate the faulty line.

Analyze an Infinite Loop:

- Debug this code:
- while (1) {
- printf("Infinite loop\n");
- }
- Use `Ctrl+C` to interrupt.
- Inspect the loop's condition with `print`.

Resolve Memory Leaks (Valgrind Integration):

- Debug memory leaks with:
- valgrind --leak-check=full ./program

Practice Problems

Debug this program using GDB:

```
#include <stdio.h>
int main() {
int x = 10, y = 0;
printf("%d\n", x / y); // Intentional division by zero
return 0;
}
```

- Identify the crash location.
- Fix the issue using GDB.

Set breakpoints and step through the following code:

```
int sum(int a, int b) { return a + b; }
int main() {
int x = 5, y = 10;
int result = sum(x, y);
printf("Result: %d\n", result);
return 0;
}
```

Useful Resources

- Official GDB Documentation: https://www.gnu.org/software/gdb/documentation/
- GDB Cheatsheet: https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf
- GDB Tutorial: https://www.cs.cmu.edu/~gilpin/tutorial/

Scenario 1: Identify a Segmentation Fault

Faulty Code:

```
#include <stdio.h>
#include <stdlib.h>


int main() {
int *ptr = NULL; // Pointer not initialized
*ptr = 42; // Dereferencing NULL pointer
return 0;
}
```

Steps to Debug with GDB:

Compile with Debugging Info:
gcc -g segfault.c -o segfault

Run GDB:
gdb ./segfault

Start the Program:
run

Identify the Fault:
GDB halts with the error:
Program received signal SIGSEGV, Segmentation fault.
Use `bt` (backtrace) to find the line causing the fault:
bt

Inspect Variables:
Check the value of the pointer:
print ptr

Fix:

Initialize the pointer properly:

```
int *ptr = malloc(sizeof(int));
*ptr = 42;
free(ptr);
```

Scenario 2: Analyze an Infinite Loop

Faulty Code:

```
#include <stdio.h>


int main() {
int i = 0;
while (i != 5) { // Logical bug: 'i' isn't updated inside the loop
printf("Looping...\n");
}
return 0;
}
```

Steps to Debug with GDB:

Run the Program in GDB:
gdb ./infinite_loop
run

Interrupt Execution:
Press `Ctrl+C` to halt the program.

Inspect the Loop Condition:
Check the value of `i`:
print i

Set a Breakpoint:
Add a conditional breakpoint after the loop:
break 7 if i == 5

Fix:

Update `i` correctly in the loop:

```
while (i != 5) {
printf("Looping...\n");
i++;
}
```

Scenario 3: Debug Memory Leaks Using Valgrind

Faulty Code:

```
#include <stdlib.h>


int main() {
int *arr = malloc(10 * sizeof(int)); // Memory allocated but not freed
arr[0] = 1;
return 0;
}
```

Steps to Use Valgrind:

Compile with Debugging Info:
gcc -g memory_leak.c -o memory_leak

Run with Valgrind:
valgrind --leak-check=full ./memory_leak

Analyze Output:
Valgrind reports a memory leak:
1 blocks are definitely lost in loss record 1 of 1

Fix the Issue:
Free the memory:
free(arr);

Integrating Valgrind with GDB:

Run Valgrind with `--vgdb=yes`:
valgrind --vgdb=yes --vgdb-error=0 ./memory_leak
Attach GDB:
gdb ./memory_leak
target remote | vgdb

Scenario 4: Step Through Library Code

Code Using `strcpy()`:

```
#include <string.h>
#include <stdio.h>


int main() {
char dest[5];
strcpy(dest, "Hello, world!"); // Buffer overflow
return 0;
}
```

Steps to Debug with GDB:

Compile with Debugging Info:
gcc -g -o buffer_overflow buffer_overflow.c

Run GDB:
gdb ./buffer_overflow

Set Breakpoint in `strcpy`:
break strcpy

Run the Program:
run

Inspect Parameters:
Check the source and destination buffers:
print dest

Fix:
Use safer alternatives like `strncpy()`:

```
strncpy(dest, "Hello, world!", sizeof(dest) - 1);
dest[sizeof(dest) - 1] = '\0'; // Null-terminate
```

Debugging Dynamic Memory Issues

Problem: Debugging errors like invalid memory access, use-after-free, or double-free.

Useful GDB Commands:

Set Breakpoints on Memory Allocators:

- break malloc
- break free

Run the Program and Analyze:

- Inspect parameters passed to `malloc` or `free`.
- print size

Detect Use-After-Free:

- Use Valgrind to identify the issue:
- valgrind --tool=memcheck ./program
- Attach GDB to pinpoint the invalid access.

Customizing GDB with Aliases

Problem: Typing long or repetitive commands can be tedious.

Solution: Create aliases or macros.

Set an Alias in `.gdbinit`:

- define alias
- document alias
- This alias sets a breakpoint at main and starts the program.
- end
- alias break main

Using Aliases: Once defined, call `alias` in GDB: alias

Analyzing Stack Overflow Errors

Symptoms of Stack Overflow:

- Program crashes or stops unexpectedly.
- Common in recursive functions.

Example Code:

```
void recursive() {
recursive();
}

int main() {
recursive();
return 0;
}
```

Steps to Debug: Compile and Run in GDB:

- gdb ./stack_overflow
- run

Analyze Stack Frames: Use `bt` to check the depth of recursion.

Fix: Introduce a base case in recursion:

```
void recursive(int depth) {
if (depth == 0) return;
recursive(depth - 1);
}
```

Debugging Race Conditions

Problem: Multiple threads accessing shared data may cause unexpected behavior.

Example Code:

```
#include <pthread.h>
int counter = 0;
void *increment(void *arg) {
for (int i = 0; i < 1000000; ++i) {
counter++;
}
return NULL;
}
int main() {
pthread_t t1, t2;
pthread_create(&t1, NULL, increment, NULL);
pthread_create(&t2, NULL, increment, NULL);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
printf("Counter: %d\n", counter);
return 0;
}
```

Steps to Debug:

- Set Breakpoints in the `increment` function.
- Monitor shared variables: watch counter
- Analyze thread execution with: info threads

Fix: Use thread synchronization:

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
counter++;
pthread_mutex_unlock(&lock);
```

Debugging Signal Handling

Problem: Understanding how the program reacts to signals (e.g., SIGINT, SIGSEGV).

Example Code:

```
#include <signal.h>
#include <stdio.h>


void handle_signal(int signal) {
printf("Caught signal %d\n", signal);
}

int main() {
signal(SIGINT, handle_signal);
while (1);
return 0;
}
```

Steps to Debug:

- Set a breakpoint in the signal handler:
- break handle_signal
- Run the program and send a signal:
- kill -SIGINT <pid>
- Analyze signal handling behavior.

Debugging Assembly Code

Disassembling Code:

- Command: disassemble <function>
- Example: disassemble main

Inspect Registers:

- Command: info registers

Set Breakpoints at Assembly Instructions:

- Example: break *0x401050

Single-Step Through Assembly:

- Command: si

Debugging Inline Assembly in C

Example Code:

```
#include <stdio.h>


int add(int a, int b) {
int result;
asm("addl %%ebx, %%eax;"
: "=a"(result)
: "a"(a), "b"(b));
return result;
}


int main() {
printf("Sum: %d\n", add(5, 3));
return 0;
}
```

Steps to Debug:

- Compile and Run:
- gcc -g inline_asm.c -o inline_asm
- gdb ./inline_asm
- Disassemble the `add` function:
- disassemble add
- Analyze registers and memory:
- info registers
- x/4xw $esp

Tips for Effective GDB Debugging

- Use `layout` commands for a split-screen view:
  - layout src
  - layout regs
- Regularly save session commands in a script.

- Use `info` commands for better understanding:
- `info functions`: List all functions.
- `info variables`: List all variables.
- `info breakpoints`: Manage breakpoints.

Conditional Breakpoints

Purpose:

- Break only when a specific condition is met.

Command:

- break <line/function> if <condition>

Example:

- break 25 if i == 10

Use Case:

- Debug loops or conditional errors without interrupting on every iteration.

Catching Signals

Purpose:

- Handle and inspect program behavior upon receiving signals like `SIGSEGV` or `SIGINT`.

Command:

catch signal <signal_name>

Example:

catch signal SIGSEGV

Inspect Signal Details:

info signals

Use Case:

- Debug signal handlers or find the root cause of a segmentation fault.

Watchpoints for Memory Changes

Purpose:

- Monitor memory or variable changes automatically.

Command:

- watch <variable>

Example:

- watch counter

Use Case:

- Track unexpected changes to variables or memory.

Examining Core Dumps

Purpose:

- Analyze program crashes by inspecting core dump files.

Steps:

- Enable core dumps:
  ulimit -c unlimited
- Run the program to produce a core dump on crash.
- Analyze with GDB:
  gdb <program_name> core

Use Case:

- Debug crashes that occur outside of GDB sessions.

Reverse Debugging

Purpose:

- Step backward in time to analyze previous states of the program.

Commands:

- Enable reverse debugging:
- target record
- Step backward:
- reverse-step
- Reverse continue:
- reverse-continue

Use Case:

- Trace back errors to their origin.

Debugging Multi-Threaded Programs

Commands:

- List all threads:
  info threads
- Switch to a specific thread:
  thread <thread_number>
- Monitor thread-specific breakpoints:
  break <function_name> thread <thread_number>

Use Case:

- Debug race conditions or thread-specific logic.

Debugging Optimized Code

Challenges:

- Variables may be optimized out or have unexpected values.
- Code may not align with source due to optimization.

Commands to Handle Optimization:

- Disable compiler optimization during compilation:
- gcc -g -O0 program.c
- Inspect register and memory directly:
- info registers
- x/4xw <address>

Automating Debugging with Python Scripts

Purpose:

- Extend GDB functionality using Python.

Steps:

- Write a Python script (`script.py`):

```
class MyBreakpoint(gdb.Breakpoint):
def stop(self):
gdb.execute("print variable")
return False # Continue execution

MyBreakpoint("main")
```

- Load the script in GDB:

```
source script.py
```

Debugging Executables Without Source Code

Commands:

- Disassemble functions:
- disassemble <function_name>
- View assembly instructions:
- x/10i <address>
- Inspect symbol table:
- info functions
- info variables

Use Case:

- Analyze third-party binaries or debug stripped executables.

Using TUI Mode in GDB

Purpose:

- Visualize source code alongside debugging information.

Commands:

- Enable TUI mode:
  tui enable
  layout src
- Switch layouts:
  layout regs
  layout asm

Use Case:

- Enhance debugging experience with a graphical view.

Advanced TUI Features

Enable TUI Mode:
tui enable

Switch Layouts:
layout src # View source code
layout asm # View assembly
layout regs # View registers

Use Case: Provide a visual debugging environment in terminal sessions.

Useful GDB Shortcuts

Shortcut Commands:

- r - Run program.
- bt - Show backtrace.
- n - Execute next line (skip function calls).
- s - Step into a function.
- c - Continue execution.
- p - Print variable value.
- q - Quit GDB.

Use Case: Increase debugging efficiency by minimizing keystrokes.

Identifying Heap Corruption

Problem:

- Heap corruption can cause unpredictable behavior.

Steps to Debug:

- Set breakpoints on memory management functions:
- break malloc
- break free
- Watch memory regions:
- watch *(int*)0xdeadbeef # Replace with the suspected address
- Use Valgrind for detailed analysis:
- valgrind --tool=memcheck ./program
- Inspect heap state in GDB:
- info proc mappings

Use Case:

- Debugging invalid memory writes or out-of-bounds access.

Using GDB to Analyze Stack Traces

Steps:

- Set a breakpoint to catch segmentation faults:
  catch signal SIGSEGV
- When the program halts, view the stack trace:
  bt
- Dive into specific stack frames:
  frame <number>
- Print local variables for a frame:
  info locals

Writing Python Scripts in GDB

Steps:

- Create a Python script to automate tasks:
- import gdb

- class WatchMalloc(gdb.Breakpoint):
- def stop(self):
- gdb.execute("bt")
- return True

- WatchMalloc("malloc")
- Load the script in GDB:
- source watch_malloc.py

Use Case:

- Automatically analyze backtraces or monitor specific conditions.

Python Expressions in GDB

- **Evaluate Python Code Directly:**
  python print(gdb.execute("info registers", to_string=True))
- **Use Python for Complex Conditions:**
  python if gdb.parse_and_eval("x") > 10:
  gdb.execute("continue")

Disassembling Code

Disassemble Entire Program:

- disassemble main

Set Breakpoints in Assembly Code:

- break *0x401050

iew Symbol Information:

- info functions
- info variables

Use Case:

- Analyze third-party or production binaries without source code.

Adding Debug Symbols to Stripped Binaries

Steps:

- Locate debug symbols:
  locate <binary>.debug
- Attach symbols to the binary:
  gdb ./program --sym ./program.debug

Use Case:

- Restore debugging capabilities for stripped executables.

Enabling Reverse Debugging

Steps:

- Record program execution:
- target record
- Step backward:
- reverse-step
- Reverse continue execution:
- reverse-continue

Use Case:

- Find where a program deviated from expected behavior.

Practical Example of Reverse Debugging

Code Example:

```
int main() {
int x = 0;
for (int i = 0; i < 5; ++i) {
x += i;
}
return x;
}
```

Steps to Debug:

- Run program in reverse:
  target record
- Step backward to see previous values:
  reverse-step

Synchronizing Thread Execution

List Threads:

- info threads

Switch Between Threads:

- thread <thread_number>

Set Thread-Specific Breakpoints:

- break <function> thread <thread_number>

Debugging Deadlocks

Steps:

- Identify blocked threads:
  info threads
- Analyze thread state:
  thread <thread_number>
  bt

Use Case:

- Detect and resolve deadlock scenarios in multi-threaded programs.

GDB + Valgrind

Purpose:

- Use Valgrind for memory error detection and GDB for in-depth analysis.

Steps:

- Run Valgrind with GDB support:
- valgrind --vgdb=yes --vgdb-error=0 ./program
- Attach GDB:
- gdb ./program
- target remote | vgdb

GDB + Perf

Purpose:

- Profile program performance using Perf and debug hotspots with GDB.

Steps:

- Profile with Perf:
  perf record ./program
- Analyze performance data:
  perf report
- Debug hotspots:
  perf annotate

Problem 1: Debugging Segmentation Fault in a Recursive Function

Scenario:

The following program crashes with a segmentation fault. Use GDB to find the root cause.

```
#include <stdio.h>


void recurse(int count) {
int arr[1000];
printf("Recursion count: %d\n", count);
recurse(count + 1); // Recursive call
}


int main() {
recurse(0);
return 0;
}
```

Objective:

- Determine why the program crashes.
- Suggest a fix to prevent the crash.
- Use GDB commands like `bt`, `info locals`, and `disassemble`.

Problem 2: Memory Leak Analysis
Scenario:
The following program runs without crashing but leaks memory. Use GDB to identify the memory leak.

```
#include <stdlib.h>


void allocate_memory() {
int *arr = (int *)malloc(100 * sizeof(int));
for (int i = 0; i < 100; ++i) {
arr[i] = i;
}
// Forgot to free memory
}


int main() {
for (int i = 0; i < 10; ++i) {
allocate_memory();
}
return 0;
}
```

Objective:

- Use GDB with Valgrind to identify the memory leak.
- Suggest modifications to fix the program.

Problem 3: Race Condition in Multi-Threaded Code
Scenario:
Debug the race condition in the following code using GDB.

```
#include <stdio.h>
#include <pthread.h>


int counter = 0;


void* increment(void* arg) {
for (int i = 0; i < 100000; ++i) {
counter++;
}
return NULL;
}


int main() {
pthread_t t1, t2;
pthread_create(&t1, NULL, increment, NULL);
pthread_create(&t2, NULL, increment, NULL);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
printf("Counter: %d\n", counter);
return 0;
}
```

Objective:

- Use GDB to observe thread execution and analyze the problem.
- Implement a solution using synchronization mechanisms (e.g., mutex).

Problem 4: Debugging with Watchpoints

Scenario:

The value of `x` is unexpectedly modified. Use a GDB watchpoint to find the line responsible.

```
#include <stdio.h>

int x = 10;

void modify_variable() {
int y = 20;
x = y + 5; // Suspect modification
}

void another_function() {
x += 10;
}

int main() {
printf("Initial value of x: %d\n", x);
modify_variable();
another_function();
printf("Final value of x: %d\n", x);
return 0;
}
```

Objective:

- Use `watch x` in GDB to track where `x` is modified.
- Report the offending line and fix the code.

Problem 5: Reverse Debugging

Scenario:

Debug the following program using reverse debugging to identify where the incorrect value of `result` is computed.

```
#include <stdio.h>


int compute(int x, int y) {
int result = x * y;
if (result > 50) {
result -= 20;
} else {
result += 10;
}
return result;
}


int main() {
int a = 5, b = 11;
int result = compute(a, b);
printf("Result: %d\n", result);
return 0;
}
```

Objective:

- Run the program in reverse mode using GDB.
- Identify where the incorrect value of `result` originates.

Problem 6: Debugging an Infinite Loop

Scenario:

The following program enters an infinite loop. Use GDB to identify the issue and fix it.

```
#include <stdio.h>


int main() {
int i = 0;
while (i < 10) {
printf("i = %d\n", i);
if (i == 5) {
continue; // Problem here
}
i++;
}
return 0;
}
```

Objective:

- Use breakpoints and `next` to trace program execution.
- Find the logic error and correct it.

Problem 7: Segfault in Dynamically Allocated Array

Scenario:

The following program crashes with a segmentation fault. Debug it with GDB to fix the problem.

```
#include <stdlib.h>
#include <stdio.h>


int main() {
int *arr = (int *)malloc(5 * sizeof(int));
for (int i = 0; i <= 5; ++i) { // Off-by-one error
arr[i] = i;
}
printf("Last element: %d\n", arr[5]);
free(arr);
return 0;
}
```

Objective:

- Use `bt` and `info locals` to pinpoint the problem.
- Fix the off-by-one error.

Problem 8: Debugging a Program with TUI

Scenario:

Debug the following program using GDB's TUI mode to step through each line visually.

```
#include <stdio.h>


int factorial(int n) {
if (n == 0) return 1;
return n * factorial(n - 1);
}


int main() {
int n = 5;
printf("Factorial of %d is %d\n", n, factorial(n));
return 0;
}
```

Objective:

- Use `layout src` to view the source code in TUI mode.
- Step through each function call and ensure correctness.