ASSIGNMENT 2

Software Engineering Lab

Member	Enrollment No
Ramesh Chandra Soren	2022CSB086
Moodu Roopa	2022CSB087
Jitendra Kumar Dodwadiya	2022CSB089
Deep Sutariya	2022CSB090

Short Note on GNU Debugger (GDB)

GNU Debugger (GDB) is a command-line tool in Linux for debugging programs written in C, C++, and other languages.

It allows developers to control program execution, set breakpoints, inspect variables, analyze memory, and diagnose crashes.

GDB supports multi-threaded debugging, core dump analysis, and attaching to running processes.

Key features include breakpoint management, stack trace inspection, watchpoints, and conditional debugging.

By stepping through code line-by-line and evaluating runtime states, GDB helps identify logical errors and resolve complex bugs.

2. Availability of GDB Commands for Given Concepts

Concept	GDB Command Available (YES/NO)
Running a program	YES
Loading symbol table	YES
Setting a breakpoint	YES
Listing variables and examining their values	YES
Printing content of an array or contiguous memory	YES
Printing function arguments	YES
Next, Continue, Set command	YES
Single stepping into a function	YES
Listing all breakpoints	YES
Ignoring a breakpoint for N occurrences	YES
Enable/disable a breakpoint	YES
Break condition and command	YES
Examining stack trace	YES
Examining stack trace for multi-threaded programs	YES
Core file debugging	YES
Debugging an already running program	YES
Watchpoint	YES

1. Running a Program

Running a program in a debugger allows controlled execution for analysis.

In GDB, the program is loaded using the `gdb <executable>` command.

It can be executed using `run` (or `r`). Arguments can be passed with `run <args>`. If the program crashes, the debugger halts execution at the faulting instruction, enabling post-mortem analysis.

Running inside a debugger ensures that breakpoints, watchpoints, and other debugging tools are available during execution.

If a program receives input, redirection (`run < input.txt`) or interactive input can be used for debugging different execution scenarios efficiently.

2. Loading Symbol Table

The symbol table contains function names, variable names, and their memory addresses.

In GDB, debugging an optimized or stripped binary might lack symbols, making debugging harder.

Compiling with `-g` (e.g., `gcc -g program.c -o program`) preserves debugging symbols.

The `info functions`, `info variables`, and `info types` commands list available symbols.

External libraries may require `set auto-load safe-path /path/to/libs` to load symbols.

Loading symbols improves backtrace quality, variable inspection, and function calls tracking.

If symbols are missing, only raw memory addresses are available instead of human-readable function or variable names.

3. Setting a Breakpoint

A breakpoint halts execution at a specific location for debugging.

In GDB, `break <line>` (or `b <line>`) sets a breakpoint at a line number.

`break <function>` halts when a function is entered.

`break <filename>:<line>` breaks at a specific file and line.

Conditional breakpoints (`break <line> if x==5`) trigger only when conditions are met.

The `info breakpoints` command lists active breakpoints.

Using breakpoints efficiently helps in stepping through execution without running the entire program, making it easier to isolate issues and inspect variables at critical points.

4. Listing Variables and Examining Their Values

While debugging, inspecting variable values helps in understanding program state.

In GDB, `print <variable>` (or `p <variable>`) prints a variable's value.

`info locals` lists all local variables in the current scope.

`info variables` displays global variables. `whatis <variable>` shows type information.

Using `display <variable>` keeps showing the variable's value after every step. Pointers can be dereferenced with `*ptr`.

For structures, `print var.member` accesses individual fields. Incorrect values often indicate logic errors or memory corruption.

Understanding variable values at different execution points helps in diagnosing issues like buffer overflows or uninitialized variables.

5. Printing Content of an Array or Contiguous Memory

Arrays and memory blocks can be printed using GDB commands like `print array`, which prints all elements for small arrays.

For larger ones, `print array[0]@size` shows `size` elements.

The `x` (examine) command inspects memory in various formats: `x/10d &array` shows 10 integers, `x/10x &array` displays 10 hexadecimal values.

The `x/s` command prints strings.

If memory corruption occurs, unexpected values may appear.

Debugging large arrays efficiently involves iterating over indices or using scripts within GDB to analyze patterns.

Understanding memory layout helps in catching segmentation faults or off-by-one errors.

6. Printing Function Arguments

Function arguments define program flow and logic.

In GDB, `info args` lists function parameters inside a stack frame.

`frame <n>` switches to a specific function frame to inspect arguments.

`print arg_name` directly prints the argument value.

When debugging crashes, verifying function arguments helps in identifying incorrect parameter passing.

If a function modifies arguments, examining their values at different points can reveal unintended mutations.

Using breakpoints on function entry with `break function` and then checking arguments with `print` helps in analyzing function behavior across different calls.

7. Next, Continue, Set Command

The next (`n`) command executes the next line without stepping into functions. The continue (`c`) command resumes execution until the next breakpoint.

The set command modifies variables during execution (`set var x = 10`).

`next` is useful for skipping over function calls, while `step` is used for diving into function calls.

Using `continue` efficiently helps in running until an issue appears, while `set` assists in simulating different scenarios without modifying source code.

These commands streamline debugging by allowing targeted execution control.

8. Single Stepping into Function

The `step` (`s`) command in GDB executes the next instruction and steps into function calls.

Unlike `next`, which executes a function as a single step, `step` enters the function, allowing examination of function internals.

If debugging a recursive function or deep call stack, stepping into every function may slow debugging.

`finish` resumes execution until the current function returns.

Stepping into library functions can be avoided using `skip` or breakpoints on user-defined functions.

This method is useful for tracking unexpected behavior in critical functions.

9. Listing All Breakpoints

Breakpoints can be managed using `info breakpoints`, which lists all active breakpoints along with their numbers, addresses, and conditions.

The `delete <breakpoint_number>` command removes specific breakpoints, while `clear` removes breakpoints at a line or function.

The `disable <breakpoint_number>` command keeps a breakpoint but prevents it from stopping execution until re-enabled.

Organizing breakpoints helps in debugging specific sections without repeatedly setting them up.

10. Ignoring a Breakpoint for N Occurrences

Ignoring a breakpoint for `N` occurrences prevents stopping execution until it has been hit `N` times.

The `ignore <breakpoint_number> <N>` command ensures the program runs normally until the Nth occurrence.

This is useful for skipping repeated function calls and focusing on later occurrences, such as debugging the 100th loop iteration instead of stopping at each iteration.

11. Enable/Disable a Breakpoint

Breakpoints can be disabled using `disable <breakpoint_number>`, preventing execution from stopping.

`enable <breakpoint_number>` reactivates it.

This is useful when certain breakpoints are temporarily not needed but should be retained for later debugging without deleting and recreating them.

12. Break Condition and Command

Conditional breakpoints (`break <line> if x == 5`) stop execution only when conditions are met.

The `commands <breakpoint_number>` feature allows executing debugger commands when a breakpoint is hit, automating variable inspection or logging values before resuming execution.

This enhances debugging efficiency by reducing manual interaction.

13. Examining Stack Trace

A stack trace shows function calls leading to a breakpoint or crash.

The `backtrace` (`bt`) command lists function calls in order, with the most recent call at the top.

Each frame contains function names, arguments, and source file locations. `frame <n>` switches to a specific stack frame for inspection.

Stack traces are critical in debugging segmentation faults, infinite recursion, and unexpected program behavior.

14. Examining Stack Trace for Multi-threaded Program

For multi-threaded programs, `thread apply all bt` shows stack traces for all threads.

The `info threads` command lists active threads, and `thread <id>` switches to a specific thread.

Debugging concurrency issues requires checking stack traces across multiple threads to diagnose race conditions, deadlocks, and synchronization errors.

15. Core File Debugging

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

Debugging a core file using `gdb <executable> core` allows post-mortem analysis.

`bt` retrieves the stack trace, and `info registers` checks CPU state at the time of failure.

This is useful for debugging crashes that occur in production but cannot be reproduced interactively.

16. Debugging an Already Running Program

Attaching to a running process using `gdb -p <PID>` allows debugging without restarting the program.

`detach` safely detaches the debugger without stopping execution.

17. Watchpoint

A watchpoint monitors memory changes.

`watch <variable>` stops execution when a variable's value changes.