GDB: Concepts and Command Availability

Software Engineering Lab (CS3273)
Assignment 2

Authors:
Uttam Mahata (2022CSB104)
Abhilash Kumar (2022CSB105)
Aryan Yadav (2022CSB106)
Harsh Raj Sahu (2022CSB107)

February 10, 2025

1 GDB: GNU Debugger

GDB (GNU Debugger) is a versatile debugging tool that helps developers track and fix program bugs. It supports inspection and modification of program state during execution through commands like 'break' (sets breakpoints), 'run' (starts program), 'next' (steps over), 'step' (steps into), and 'print' (shows variable values). GDB works with languages like C, C++, and Fortran, and can debug both local and remote programs. Its features include conditional breakpoints, watchpoints, and call stack examination. Many IDEs use GDB as their backend debugger while providing a graphical interface.

2 GDB Concept Availability

GDB Concept	**GDB Command Available**
Running a program	YES
Loading symbol table	YES
Setting a break-point	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 function	YES
Listing all break points	YES
Ignoring a break-point for N occurrence	YES
Enable/disable a break-point	YES
Break condition and Command	YES
Examining stack trace	YES
Examining stack trace for multi-threaded program	YES
Core file debugging	YES
Debugging of an already running program	YES
Watchpoint	YES

Table 1: GDB Concepts and Command Availability

3 Explanation of Concepts and Commands

1. **Running a program**: Starts the program you want to debug. The run command (or its shorthand r) in GDB is the fundamental command that initiates the execution of the program you're intending to debug. It essentially instructs GDB to start the program, loading it into memory and beginning the program's process. It's the starting pistol for your debugging session. **Loading:** GDB loads the executable into memory. This includes the program's code, data segments, and other necessary components. It ensures the program is ready to execute.

   Initialization: The operating system and GDB collaborate to initialize the program's execution environment. This includes:

   - Setting up the program's stack for function calls and local variables.
   - Loading any dynamic libraries (shared objects) that the program depends on.
   - Preparing the program's memory space.
   - Setting the initial program counter (the instruction pointer) to the program's entry point (usually the start of the `main` function).

   Execution Begins: Once the initialization is complete, GDB instructs the operating system to start executing the program's instructions. The program then proceeds normally, following its defined logic and instructions.

   - *Command*: `run` or `r`
   - *Example*: `(gdb) run`

2. **Loading symbol table**: GDB needs debug info (symbols) to understand names. The symbol table is crucial for effective debugging, bridging the gap between machine-level instructions and human-readable names.

   Definition:

   The symbol table is a data structure embedded within (or associated with) the executable, containing information about function names, variables, data types, and their corresponding memory addresses. It maps names to addresses.

   Elaboration:

   When compiling with debugging information (e.g., `-g` flag), the compiler generates the symbol table, including:

   - Function Names (e.g., `main`, `calculate_average`).
   - Variable Names (global and local).
   - Data Types (`int`, `float`, `char *`, structures).
   - Line Numbers (source code line to machine code mapping).
   - File Names.
   - Structure and Class Definitions.

   Why GDB Needs the Symbol Table:

   - **Name Resolution:** Translates names to memory addresses.
   - **Breakpoint Setting:** Determines memory addresses for breakpoints.
   - **Stack Traces:** Generates readable stack traces with function names.
   - **Variable Inspection:** Allows inspecting variable values by name.
   - **Source Code Display:** Displays source code using line number mappings.

   Commands for Loading the Symbol Table:

- `file program-name`: Loads the symbol table from the executable.
- `symbol-file filename`: Loads the symbol table from a separate file.

Important Considerations:

- Compile with `-g` for debugging information.
- Avoid stripped executables (they lack symbol tables).
- Ensure symbol file consistency with the executable version.
- For dynamic libraries, GDB usually loads symbols automatically, but `set solib-search-path` might be needed.

In essence, loading the symbol table enables GDB to ”understand” your code, making debugging easier and more effective.

3. **Setting a Breakpoint:** Pauses program execution at specific locations.

   Definition: A breakpoint is a marker set in the program's code that tells the debugger to temporarily halt execution when that point is reached. This allows you to inspect the program's state (variables, stack, etc.) at a particular point in time.

   Elaboration: Breakpoints are essential for controlling the flow of execution during debugging. They let you focus on specific areas of interest in your code. When the program reaches a breakpoint, GDB pauses execution, giving you control to examine variables, step through code, or continue execution.

   - **Command:** `break` or `b`
   - **Example:**
     - `(gdb) break main`: Sets a breakpoint at the beginning of the `main` function.
     - `(gdb) break 10`: Sets a breakpoint at line 10 of the current source file.
     - `(gdb) break myfunction`: Sets a breakpoint at the beginning of the function `myfunction`.
     - `(gdb) break filename.c:25`: Sets a breakpoint at line 25 of the file `filename.c`.
     - `(gdb) break *0x4000`: Sets a breakpoint at the memory address `0x4000`. (Less common; requires knowing the exact address).

4. **Listing Variables and Examining Their Values:** Shows the current values of program variables.

   Definition: Examining variables involves inspecting the contents of memory locations associated with variable names. This allows you to see the state of your program's data as it executes.

   Elaboration: Knowing the values of variables at different points in execution is fundamental to understanding program behavior. GDB provides commands to display the values of variables, expressions, and even the contents of memory locations.

   - **Command:** `print` or `p`
   - **Example:**
     - `(gdb) print x`: Prints the current value of the variable `x`.
     - `(gdb) print y + z`: Prints the value of the expression `y + z`.
     - `(gdb) print *ptr`: Prints the value pointed to by the pointer `ptr`.
     - `(gdb) print my_struct.member`: Prints the value of the member `member` of the structure `my_struct`.
     - `(gdb) display x`: Makes GDB automatically print the value of `x` every time the program stops.

5. **Printing Content of an Array or Contiguous Memory:** Allows you to view memory blocks.

   Definition: This feature lets you inspect a contiguous region of memory, typically representing an array or a block of dynamically allocated memory.

   Elaboration: When debugging, it's often necessary to view the contents of arrays or other memory blocks to ensure that data is being stored and manipulated correctly. GDB provides ways to specify the starting address and the number of elements to display.

- **Command:** `print` or `x`
- **Example:**
  - `(gdb) print myarray@10`: Prints the first 10 elements of the array `myarray`.
  - `(gdb) x/10dw myarray`: Examines 10 double words (8 bytes each) starting at the address of `myarray` and displays them in decimal format. The `x` command is more versatile.
  - `(gdb) x/10i 0x4000`: Examines 10 machine instructions starting at memory address `0x4000`. This can be useful for disassembling code.
  - `(gdb) x/10s string_ptr`: Examines 10 null-terminated strings starting at address in `string_ptr`.

6. **Printing Function Arguments**

   Definition: This command is used to inspect the input values passed to a function at the point where execution has been paused (e.g., at a breakpoint). It's vital for verifying that a function is receiving the correct data and helps diagnose issues arising from incorrect or unexpected input.

   Elaboration:

   - **Purpose:**
     - Confirming the inputs that a function is receiving when it's called.
     - Understanding if any transformations or calculations performed on the arguments before the function call are producing the expected results.
     - Diagnosing issues related to function parameters having incorrect or unexpected values, leading to faulty or incorrect execution behavior.
   - **Commands:**
     - `info args`: Displays all arguments of the current function and their values in a concise format. It lists each parameter name along with its current value.
     - `print arg_name`: Allows printing the value of a specific argument. This is particularly useful when you have many arguments and only need to focus on one or a few. You can print the value of expressions such as the members of struct to evaluate the complete argument for debugging.
   - **Practical Usage:**
     - Set a breakpoint at the start of a function: `(gdb) break function_name`
     - Run the program: `(gdb) run`
     - Once the program hits the breakpoint, use `info args` to inspect all function arguments or `print arg_name` to view the value of a specific argument.

7. **Next, Continue, Set Command**

   Definition: These are essential commands to control the execution of a program during debugging.

   - `next` or `n`: Executes the current source line without stepping into function calls.
   - `continue` or `c`: Resumes program execution until a breakpoint or error is encountered.
   - `set`: Changes the value of a variable during a debugging session.

   Elaboration:

   - **next** or `n`: Executes the current line of code, stepping over any function calls. This means that if the current line calls another function, the debugger will execute that function completely, and then stop at the next line of code in the *current* function. This is ideal when you're not interested in debugging the called function.
     - Example: `(gdb) next` - Executes the current line.
   - **continue** or `c`: Resumes the execution of the program until it encounters the next breakpoint, or until the program terminates. This is useful when you want to let the program run at full speed, only stopping at specific locations that you have marked with breakpoints.
     - Example: `(gdb) continue` - Resumes program execution.

- **set**: Allows you to change the value of a variable while the program is running. This is useful for testing different scenarios, for correcting errors, or for examining what happens when a variable takes on a different value.
- Example: `(gdb) set variable_name = new_value` - Sets the value of variable ‘variable$_name‘to‘new_value$‘.

8. **Single Stepping Into Function**

   Definition: Single stepping allows executing one line of source code at a time. Specifically, ”stepping into” a function means following the execution flow *into* a function call on the current line.

   Elaboration:

   - **Functionality:** When the current line contains a call to a function, the `step` (or `s`) command causes GDB to enter that function, pausing execution at the first line of code within the called function.
   - **Difference with `next`** The major distinction between ‘step‘ and ‘next‘ lies in how they handle function calls. While ‘next‘ executes the entire function call without pausing, ‘step‘ delves into it.
   - **Usage Scenario:** A program exhibiting unexpected behavior inside a particular function makes ‘step‘ the command to use to check each line of code. If the program consists of many functions that call each other, step gives a way to ”dive in” and inspect each function.

9. **Listing All Breakpoints**

   Definition: Provides a comprehensive view of all active breakpoints currently set within a GDB debugging session. It serves as a tool for reviewing and managing your breakpoints during debugging.

   Elaboration:

   - **Purpose:** The `info breakpoints` (or `i b`) command displays a list of all active breakpoints.
   - **Included information:** For each breakpoint, it typically provides the following information:
     - Breakpoint number: A unique identifier for the breakpoint.
     - Type: The type of breakpoint (e.g., line breakpoint, function breakpoint).
     - Disposition: Whether the breakpoint is enabled or disabled.
     - Address: The memory address at which the breakpoint is set.
     - Location: The source file and line number or function name.
     - Condition: The condition that must be true for the breakpoint to be hit.
     - Ignore count: The number of times the breakpoint will be ignored before stopping the program.
   - **Utility:** A program with many functions calling each other can easily have many breakpoints. Calling `info breakpoints` provides a quick snapshot to see how the execution will proceed.

10. **Ignoring a Breakpoint for N Occurrences**

    Definition: This command allows you to temporarily disable a breakpoint for a certain number of hits. The program will resume without stopping until the breakpoint would have been hit this many times, at which time the program execution will stop.

    Elaboration:

    - **Purpose:** To skip over a breakpoint for a specified number of times it is hit. This can be useful when debugging loops or recursive functions, where you want to examine the program state after a certain number of iterations or recursive calls.
    - **Command:** `ignore bnum count`
    - **How it works:** The ‘ignore bnum count‘ command tells GDB to skip the breakpoint number ‘bnum‘ for the next ‘count‘ times that it is encountered. After these skips, the breakpoint functions normally, stopping the program as expected. It’s useful when you know that the program will hit a certain breakpoint many times, but you’re only interested in examining what happens after a certain number of hits.
    - **Scenarios:**

  - When stepping through a loop or function call, ignoring the first few times it is hit, but stopping at a later time.
  - When focusing on a specific edge case of a loop, but stepping over earlier values that the code operates on.

11. **Enable/Disable a Breakpoint**

    Definition: This functionality allows you to activate or deactivate existing breakpoints during a debugging session, offering flexibility in controlling the flow of execution.

    Elaboration:

    - **Purpose:** Enable and disable commands provide a way to temporarily bypass a breakpoint without removing its settings. This can be useful when you know you want to skip a certain breakpoint for a while but don't want to have to recreate it later.
    - **Commands:**
      - `enable bnum`: Activates the breakpoint numbered 'bnum'. When a breakpoint is enabled, the program will halt execution when it reaches the specified location.
      - `disable bnum`: Deactivates the breakpoint numbered 'bnum'. When a breakpoint is disabled, the program will ignore it and continue execution as if the breakpoint were not there.
    - **Usage Scenario:** When debugging a loop, the breakpoint could be disabled after n occurrences but re-enabled when a user wants to step into it.

12. **Break Condition and Command**

    Definition: This feature enables the creation of more sophisticated breakpoints that trigger only when specific conditions are met, and optionally execute a list of commands when triggered.

    Elaboration:

    - **Purpose:** To set breakpoints that are triggered only when a certain condition is true. This allows you to stop the program only when a specific condition is met. To perform an action (e.g. printing a variable) every time the code reaches the breakpoint.
    - **Commands:**
      - `condition bnum expression`: Sets a condition on breakpoint number 'bnum'. The breakpoint will only trigger when 'expression' evaluates to true.
      - `commands bnum ... end`: Defines a sequence of GDB commands to be executed automatically when breakpoint number 'bnum' is hit. The commands are enclosed between the 'commands bnum' and 'end' keywords.
    - **Usage Scenario:** Printing the value of a variable x inside a function if the incoming argument is greater than n to isolate the reason for the incorrect behaviour, by setting `condition bnum expression` for an incoming argument being less than n.

13. **Examining Stack Trace**

    Definition: A stack trace, also known as a backtrace, provides a hierarchical view of the function calls that led to the current point of execution in the program. It is a vital debugging tool for understanding the sequence of function calls and pinpointing the origin of errors.

    Elaboration:

    - **Purpose:** To trace back the sequence of function calls that lead to a current point. This allows you to see what functions have been called in what order, and what the relationships are between them.
    - **Command:** `backtrace` or `bt`
    - **Content of Stack Trace:** The stack trace typically displays each function call along with its arguments and the source file and line number where the call was made. The most recent function call is at the top of the trace, and the initial function call is at the bottom.

- **Usage Scenario:** When the program crashes or exhibits unexpected behaviour at a certain point, this allows a debugger to track where the execution has come from.

14. **Examining Stack Trace for Multi-Threaded Program**

    Definition: In a multi-threaded program, examining the stack trace of each thread is essential for understanding the program's overall state and identifying issues related to concurrency, synchronization, or data sharing.

    Elaboration:

    - **Purpose:** Obtaining a backtrace of all threads in the program. The thread's stack trace offers a way to visualise how the function is calling itself, and what the input arguments are.
    - **Commands:**
      - `thread apply all bt`: Generates a stack trace for *all* threads in the program. This is often the first command to run in a multi-threaded debugging session.
      - `thread n bt`: Generates a stack trace for thread number 'n'. You need to identify 'n' beforehand, which can be done with `info threads`.
    - **Usage Scenario:** When debugging a multi-threaded application, it could be helpful to examine the call stack of all threads for identifying how the threads interact with each other, as well as finding race conditions and deadlocks.

15. **Core File Debugging**

    Definition: A core file is a memory dump of a program's state at the time of a crash or abnormal termination. Debugging with a core file allows you to analyze the state of the program at the point of failure, even if the program is no longer running.

    Elaboration:

    - **Purpose:** Examining the core file is vital for gaining insights into program crashes.
    - **Command:** `gdb program-name core-file`
    - **Core File Contents:** It shows variable values, stack contents, and other information relevant to assessing why the program failed.
    - **Usage Scenario:** Analyzing a program crash that can't be reproduced for identifying the root cause.

16. **Debugging of an Already Running Program**

    Definition: This allows you to attach GDB to a process that is already running, enabling you to examine and control the program's execution in real-time without restarting it.

    Elaboration:

    - **Purpose:** Examining processes running on the system for debugging purposes, when you didn't launch the process itself.
    - **Command:** `attach pid`
    - **Usage Scenario:** Attach to a production process for debugging purposes.

17. **Watchpoint**

    Definition: A watchpoint is a special type of breakpoint that triggers when the value of a specified variable or memory location is accessed (read or written). It allows you to track data changes during program execution.

    Elaboration:

    - **Purpose:** To stop execution when a variable's value is changed (or accessed), or when the data in a memory location is read, written, or accessed.
    - **Commands:**

  - `watch variable`: Triggers when the value of ‘variable‘ is either read or written.
  - `rwatch variable`: Triggers only when the value of ‘variable‘ is *read*.
  - `awatch variable`: Triggers only when the value of ‘variable‘ is *accessed*.

- **Usage Scenario:** Finding when a variable is changed incorrectly. This is the core reason for setting a watchpoint.