Debugging with gdb

VIVEN

Embedded Academy

Debugging

- Debugging is a process of locating and fixing errors (known as bugs) in a computer program.
- Debugging is a necessary process in almost any new software.
- Debugging tools identify coding errors at various stages of development.
- gdb is one of the debugging tool used in linux.

Compiling program for gdb

- To debug a program effectively, we need to generate debugging information when we compile it.
- This debugging information is stored in the object file. It describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

Compiling program for gdb

- To compile the program to support debugging use “–g” option.
- The command is as follows:

```
gcc –g  prog.c  -o prog
```

Invoking gdb

VIVEN Embedded Academy

- gdb

  starting gdb

- gdb prog

  invoking gdb with an executable

- gdb - - silent prog

  invoking gdb in silent mode, to stop printing the front material , which describes gdb.

- gdb - -help

  displays all available options

Quitting GDB

VIVEN Embedded Academy

- quit or q
    - To exit GDB

Tips on Gdb

VIVEN Embedded Academy

- Most of the commands in GDB can be issued by simply giving the starting letter.
  - → For ex:

    '*b*' can be used instead of '*break*'.

- A blank line as input to GDB means to repeat the previous command.

Tips on Gdb

VIVEN Embedded Academy

- Any text from a # to the end of the line is a comment.
- Press the TAB key whenever you want GDB to fill out the rest of a word.

Running the program

- run (r)
  - to start program under GDB.
- run arg1 arg2
  - to run with command line args.
- run > outfile
  - diverting output to the file ‘outfile’

Working Directory

VIVEN Embedded Academy

- By default the working directory will be the present working directory.

- We can specify a new directory with the cd command.
  - cd directory
  - pwd
    - **prints the GDB working directory.**

Passing Arguments

VIVEN Embedded Academy

- You can also pass arguments to program by using '*set args*'
  - For ex:
    - **set args 1 2 3**
- To display args use '*show args*'

  show args

Debugging a running process

- attach

  attaches a running process to the gdb.

  Ex:

  attach 5344

  where 5344 is the process-id.

- detach

  releases a process from GDB control.

Stopping & Continuing

- Breakpoints
- Watchpoints
- Catchpoints

breakpoints

VIVEN Embedded Academy

- Breakpoints stops the program when certain point is reached.
- ‘break’ (b) command is used to set break points.

Setting Breakpoints

VIVEN Embedded Academy

- break function
  - Sets a break point at entry to function name.
- break +offset
- break -offset
  - Set a break point some number of lines forward or back from the position at which execution stopped.
- break linenum
  - Set a break point at line linenum in current sourcefile.

Setting Breakpoints

VIVEN Embedded Academy

- break filename:linenum
  - Set a break point at the linenum in source file filename.
- break filename:function
  - Set a breakpoint at entry to function function found in file filename.
- break
  - When called without any arguments, break sets a breakpoint at the next instruction to be executed.

Setting Breakpoints

VIVEN Embedded Academy

- break ... if cond
  - Set a breakpoint with a condition,stops only if the condition returns non-zero .
- info breakpoints
  - Prints a table of all breakpoints.

Watchpoints

VIVEN Embedded Academy

- Stops execution whenever the value of an expression changes.
- watch expr
  - Set a watchpoint for an expression.
  - GDB will break when expr is written and its value changes.
- rwatch expr
  - Break when expr is read by program.
- awatch expr
  - Break when expr is either read or written into by the program.

Setting Catchpoints

VIVEN Embedded Academy

- catchpoints cause the debugger to stop for certain kinds of program events.
- catch event
  - Stops when event occurs.
  - Some of the events are
    - fork
      - A call to fork.
    - exec
      - A call to exec
  - We can also run a set of commands when control reaches a breakpoint.

Delete breakpoints

VIVEN Embedded Academy

- clear
  - Delete any breakpoints at the next instruction to be executed.
- clear function
  - clear filename:function
  - Delete any breakpoints set at entry to the function.
- clear linenum

Delete breakpoints

- clear filename:linenum
  - Delete breakpoints set at a specific line.
- delete
  - Delete all breakpoints, catchpoints , watchpoints.

Continuing & Stepping(1/3)

- *Continuing* means resuming program execution until your program completes normally.
- *stepping* means executing just one line of source code, or one machine instruction.
- continue (c)
  - Resume program execution.

VIVEN
Embedded Academy

Continue & Stopping (2/3)

- next (n) [count]
  - executes next instruction & stops.

- step (s) [count]
  - executes next instruction & stops.

  - Diff b/w next & step is that, if it is a function call it proceeds until the function returns.
  - ➢ We can execute count no of instructions by giving a count.

Continue & Stopping (3/3)

- nexti (ni)
- stepi (si)
  - Both are same as next & step but the difference they execute a machine instruction.

Signals

VIVEN Embedded Academy

- We can tell GDB to perform an action when a signal rises by *handle*.

- handle *signal keyword*

  - Signal can be a number or name.
  - The keyword says what change to make.

Signals

VIVEN Embedded Academy

- keywords are:
  - nostop

    Do not stop the program.
  - stop

    Stop the program.
  - print

    Print a message when a signal happens.
  - pass/noignore

    GDB should allow the signal so that your program can handle the signal.

Signals

VIVEN Embedded Academy

- nopass/ignore

  GDB should not allow your program to see the signal.

- info signals
- info handle

  Print a table of all the kinds of signals and how GDB has been told to handle each one.

Printing Source lines

- list linenum
  - prints lines centered around *linenum* in the current source file.
- list function
  - prints lines centered around beginning of the function.
- list
- set listsize count
- show listsize

Examining Data

VIVEN Embedded Academy

- print expr
  - value of expr is printed.
- print (p) variable
- print (p) file::variable
- print (p) func::variable
  - Ex: p ‘add.c’::a
- To display automatically use display.
  - Ex display expr
- To delete display use undisplay or delete display dnums

Altering Execution

VIVEN Embedded Academy

- Assigning Variables
  - print x=4
  - changes the x value and prints.
  - set x=4
- whatis
  - helps you know about a given keyword
  - For Ex
    - whatis x   // tells type of x

jump

VIVEN Embedded Academy

- helps to go to a specific target
- jump func:line
  - jumps to a particular line in function func.
- jump linenum
  - jumps to a specific line.
- jump *addr

  Resume execution from instruction at address.

Sending a signal

- We can send a signal to program by using signal
- For ex
  - signal signal1

return

VIVEN Embedded Academy

- We can cancel execution of a function call with the return command.
- return expr

  If we give an expression its value is used as the function’s return value.

logging

- set logging on

  Enable logging
- set logging off

  Disable logging
- set logging file out.txt
  - Change the name of the current logfile, default is gdb.txt
- help
  - displays a short list of classes of commands

Profiling tools

Profiling tools

- Profiling tools help you analyze your code's performance.
- By using profiling tools we can find out some basic performance statistics, such as:
  - how often each line of code executes
  - what lines of code are actually executed
  - how much computing time each section of code uses

- Some of the profiling tools are
  - gcov
  - gprof

gcov

gcov

- The gcov utility is a coverage testing tool.
- It monitors an application under execution and identifies which source lines have been executed and which have not.
- gcov can identify the number of times a particular line has been executed.
- The gcov utility is used in conjunction with the compiler tool chain.

Compiling for gcov

- To enable coverage testing the program must be compiled with the following options.
  - gcc -Wall -fprofile-arcs -ftest-coverage cov.c
  - Creates an executable which contains additional instructions that record the number of times each line of the program is executed.

Compiling for gcov

- In order to use gcov on a program use the

“-fprofile-arcs” and

“-ftest-coverage”

options with gcc or g++.

- This tells the compiler to generate additional information needed by gcov.

Procedure for using gcov

- Example:
  - gcc sort.c -o sort -fprofile-arcs -ftest-coverage
  - Run the executable file such as

    ./sort
  - gcov sort.c
  - It creates a file with a source file name with an extension of .gcov, such as

    sort.c.gcov

  The statistics are present in this file.

gprof

gprof

- It helps in identifying how much time is spent for a particular function.
- It also finds which functions were called by a given function.

Procedure for using gprof

- gcc sort.c - sort –pg
- After given the above command run the execuatble such as

  ./sort

  A file named gmon.out will be created.

  Run the program once again to generate statistics as

  gprof a.out gmon.out > sort.gprof

  sort.gprof gives the statistics of excution details.

Dos2unix & unix2dos

- DOS files use a different set of control characters for the end of a line. To convert between DOS and Unix files, there are 2 commands:
  - dos2unix and
  - unix2dos

dos2unix

- Description:
  - This is used to convert plain text files in DOS/MAC format to UNIX format.
- Example:
  - dos2unix file.txt converts file.txt to unix format.

unix2dos

- Description:
  - This is used to convert plain text files in UNIX format to DOS/MAC format.
- Example:
  - unix2dos file.txt converts file.txt to dos format.