BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJASTHAN) – 333 031
/IS F462 Network Programming
I semester 2014-15
Lab1

AGENDA

- **Debugging**
- **Profiling**
- **Makefile**
- **Process Info**

DEBUGGING

Debugging is something that can't be avoided. Every programmer will at one point in their programming career have to debug a section of code. There are many ways to go about debugging, from printing out messages to the screen, using a debugger, or just thinking about what the program is doing and making an educated guess as to what the problem is.

Before a bug can be fixed, the source of the bug must be located. For example, with segmentation faults, it is useful to know on which line of code the seg fault is occurring. Once the line of code in question has been found, it is useful to know about the values in that method, who called the method, and why (specifically) the error is occurring. Using a debugger makes finding all of this information very simple.

> ***TRY THIS:*** Compile & run the files *prog1.c* &*prog2.c* given to you. You should get a segmentation fault in one and an infinite loop in the other. You can check and remove these error by inspection as it is quite easy to remove. Still, read the following and detect the error using gdb, as the technique is important and things will not be so easy as your codes becomes longer.
>
> **Note:** *prog2.c* takes user input on the command line.

GNU Debugger (GDB):

Gdb is a debugger for C (and C++). It allows you to do things like run the program up to a certain point then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line. It uses a command line interface.

NOTE: Use the ***help*** command on the gdb prompt to get more information on any commands discussed below.

> **Compiling for GDB:**
>
> Add a **-g** option to enable built-in debugging support (which gdb needs):
>
> **gcc [other flags] -g<source files> -o <output file>**
>
> **Starting GDB:**
>
> Start by simply running *gdb* on the prompt. This will enter the *gdb* prompt. (to quit this prompt anytime use the ***quit***command)

Loading Files in GDB:

Two ways:

1. Start gdb as ***gdb executable*** // the executable refers to the output_file you created when compiling
2. After gdb has started use the command ***file execulatble***to load the file.
   NOTE: Source file should be in the same place as executable.

Running file in GDB:

1. Use the command ***run [arg1] [arg2]…***from the gdb prompt
2. Use the command ***kill*** to end the current running program (after you have stopped it).

Starting & Stopping:

a. You can **stop execution** by sending your program UNIX symbols like SIGINT. This is done using the **Ctrl-C** key combination (when the programming is running in gdb prompt using *run* command. Your program also stops when it exits abnormally (segmentation fault,etc)
b. Use the **continue** command to restart execution of your program whenever it is stopped.
c. Use the **list** command to have gdb print out the lines of code above and below the line the program is stopped at.
d. Use the **next** and **step** commands to step through your code line-by-line (when it's stopped).

   NOTE: the next and step commands are different. On a line of code that has a function call, next will go 'over' the function call to the next line of code, while step will go 'into' the function call.

Variables:

a. Use the **print** command with a variable name as the argument to check its value.
   NOTE 1: variables names accessed using scope rules in the block the program is stopped in.
   NOTE 2: The output from the print command is always formatted $## = (value). The $## is simply a counter that keeps track of the variables you have examined.

b. Use the **set var** command with a C assignment statement as the argument. For example, to change int x to have the value 3:
   (gdb) set var x = 3

Functions:

a. From the debugger command line you can use the **call** command to call any function linked into the program. This includes your own code as well as standard library functions.
   Ex: (gdb) call abort()
b. Use the **finish** command to have a function finish executing and return to it's caller. This command also shows you what value the function returned.

Breakpoints:

Breakpoints are set points (lines or functions) in the program where gdb will stop executing the program (before the breakpoint is executed).

a. The command to set a breakpoint is **break**. If you only have one source file, you can set a breakpoint like so:

```
(gdb) break 19
Breakpoint 1 at 0x80483f8: file test.c, line 19
```

b. If you have more than one file, you must give the break command a filename as well:

```
(gdb) break test.c:19
Breakpoint 2 at 0x80483f8: file test.c, line 19
```

c. To set a breakpoint on a C function, pass its name to break.

```
(gdb) break func1
Breakpoint 3 at 0x80483ca: file test.c, line 10
```

d. Use the **info breakpoints** command to get a list of current breakpoints.
e. Use the **disable** command to disable a breakpoint. Pass the number of the breakpoint you wish to disable as an argument to this command (this number is got from the previous info command).
f. To skip a breakpoint a certain number of times, we use the **ignore** command. The ignore command takes two arguments: the breakpoint number to skip, and the number of times to skip it.

Watchpoints:

Watchpoints are similar to breakpoints. However, watchpoints are not set for functions or lines of code. Watchpoints are set on variables. When those variables are read or written, the watchpoint is triggered and program execution stops.

a. Use the **watch** command passing a variable name to set a watchpoint.
NOTE: The argument to the watch command is an expression that is evaluated. This implies that the variabel you want to set a watchpoint on must be in the current scope. So, to set a watchpoint on a non-global variable, you must have set a breakpoint that will stop your program when the variable is in scope. You set the watchpoint after the program breaks.

b. **info breakpoints** &**disable** commands work as above.

Call Stack:

a. Use the command **backtrace** to see the current call stack.
b. Use the command **frame**to change the current stack frame (to access a caller functions local variables, etc.). Pass the number of the frame you want as an argument to the command (the frame number is displayed in the above backtrace command).
c. To look at the contents of the current frame, there are 3 useful gdb commands. **info frame** displays information about the current stack frame. **info locals** displays the list of local variables and their values for the current stack frame, and **info args** displays the list of arguments.

Debug prog2.c (infinite loop):

Load and run the program in gdb. Enter some user-input followed by a new line to get into the infinite loop.

a. Press Ctrl-C to send the program a SIGINT (in the infinite loop).
b. Use the backtrace command to examine the stack.
c. Depending on when you pressed Ctrl+C, you will get different outputs. But what we really want to see is where we are in main, so switch to that frame number.
d. Check the value of c. Is it what you expected?
e. The following command is to be used if your top-most frame was not main.
    a. Now we have to find the loop. We use several iterations of the 'next' command to work our way up the call stack back to main(). The next command will exit any functions we can't debug (like C library functions). We could also use the **finish** command here.
f. Inside main(), we run the next command several more times to watch the program execute.
   NOTE: In case using next simply exits your program and you get a 'your program is not running' message, kill and load the file again. This time before running set a breakpoint at the 'printf' line (13), since that is where the programming is getting stuck.
g. Notice a pattern? The same two lines of code are executing over and over.

The lines being executed are 12 and 13. The test is always passing because the character is never changing. So the program is only reading characters until it finds an alphanumeric character, after which it never reaches the fgetc. But we always want to read the next character. Removing the 'else' on line 14 will fix the bug.

PRACTICE:

The above was a very simple program. Try debugging *prog1.c* and then try debugging the rest of the simple programs in the gdb directory.

REFERENCES:

1. http://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf
2. http://web.eecs.umich.edu/~sugih/pointers/summary.html
3. http://www.cs.cmu.edu/~gilpin/tutorial/ (for C++)

GUI: http://www.gnu.org/software/ddd/

PROFILING:

Profiling allows you to learn where your program spent its time and which functions called which other functions while it was executing. This information can show you which pieces of your program are slower than you expected, and might be candidates for rewriting to make your program execute faster. It can also tell you which functions are being called more or less often than you expected. This may help you spot bugs that had otherwise been unnoticed.

Try profiling the given biophysics.c

Compiling for profiling:

Add a **-pg** option to enable gprofprofiling support :

gcc [other flags] -pg<source files> -o <output file>

Running:

Run the program executable normally. An extra output file named gmon.out would be generated for use by gprof.

Profile Information:

Run the command***gprof executable > profile_info.txt***

The file *profile_information* will contain verbose information about the profiled data. It's divided into the **flat profile** &**call graph.**

Flat Profile:

The flat profile shows the total amount of time your program spent executing each function. Unless the `-z' option is given, functions with no apparent time spent in them, and no apparent calls to them, are not mentioned.

Note that if a function was not compiled for profiling, and didn't run long enough to show up on the program counter histogram, it will be indistinguishable from a function that was never called.

The functions are sorted by decreasing run-time spent in them. The functions `**mcount'** and `**profil'** are part of the profiling aparatus and appear in every flat profile; their time gives a measure of the amount of overhead due to profiling.

Call Graph:

The call graph shows how much time was spent in each function and its children. From this information, you can find functions that, while they themselves may not have used much time, called other functions that did use unusual amounts of time.

Useful command-line switches with gprof:

- Suppress the printing of statically(private) declared functions using –a
- Suppress verbose blurbs using –b
- Print only flat profile using –p

- **Print information related to specific function in flat profile. This can be achieved by providing the function name along with the –p. (ex: gprog –pfunc1)**
- Suppress flat profile in output using –P
- Print only call graph using –q
- **Print information related to specific function in call graph. This can be achieved by providing the function name along with the –q. (ex: gprog –qfunc1)**
- Suppress flat profile in output using –Q

References:

1. https://www.cs.utah.edu/dept/old/texinfo/as/gprof.html
2. https://www.cs.duke.edu/~ola/courses/programming/gprof.html

MAKEFILES:

Compiling your source code files can be tedious, especially when you want to include several source files and have to type the compiling command every time you want to do it.

The Make Utility

If you run **make** this program will look for a file named **makefile** in your directory, and then execute it.

If you have several makefiles, then you can execute the specific one with the command-line switch –f followed by the specific name:

make -f MyMakefile

Structure of Makefile:

The basic makefile is composed of:

target: dependencies

[tab] system command

NOTE: the makefiles require 'tabs', not spaces. Make sure your editor does not replace tabs with spaces.

Example: The makefile for compiling a simple C program would be.

all:

    gcc program.c

NOTE: the first target listed in the makefile is the default target name for *make.*If you have other targets, pass the name of the target along with *make.*

Example: This makefile has a target for cleaning all executables with the extension out.

all:

    gcc program.c

clean:

    *rm –rf *.out*

You need to run *make clean* to delete the *.out executables. Simple running *make* will only compile program.c

Dependency System:

NOTE: The above examples did not have any dependcies in the targets.

Makefile targets resolve dependencies as further target names – creating chained targets. A Makefile target will only be executed when all its dependencies have been executed.

Dependencies can also be filenames existing in the directory. In such a case, the target is only executed if any of the files has been changed/more recent than the last time the target was executed.

NOTE: There is a catch in the above description. Unless make keeps databases of targets its executes and their timestamps, there is no way it can find out when was the last executuion of a target. Since, it does not keep databases, the implication is that such targets themselves must be referring to a file existing in the directory, whose last modification time is used by make for comparison with the dependencies.

Example: Revisiting the previous example with a file target

all: program

program: program.c

    gcc program.c –o program

Now, make will only execute gcc if the source file has changed since last time the executable 'program' was built.

Example: Multiple dependencies can be written, seperated by spaces.

all: program

program: main.o library.o

    gcc main.o library.o -o program

main.o: main.c

    gcc -c main.c

library.o: library.c

    gcc -c library.c

Variables:

Makefiles supports using variables, delimited by newlines.

Example: Rewriting the above makefile with variables

#comments in makefile are started on newlines with a 'hash' character

#set the compiler

CC=gcc

#set compiler flags

CFLAGS=-c

#set dependencies for program

DEPS_PROG=main.o library.o

#use variables by putting them in brackets prefixed with a dollar sign

all: program

program: ***$(DEPS_PROGS)***

    $(CC) $(DEPS_PROGS) *-o program*

main.o: main.c

    $(CC) $(CFLAGS) main.c

library.o: library.c

    $(CC) $(CFLAGS) library.c

Using Wildchars & Macros:

Makefiles supports the use of wildcharacters to refer to groups of files as well as special macros to refer to the dependency list and target name.

Example: Rewriting the above makefile with wildchars

CC=gcc

CFLAGS=-c

DEPS_PROG=main.o library.o

all: program

program: $(DEPS_PROGS)

$(CC) $(DEPS_PROGS) -o ***$@***

%.o*: %.c*

$(CC) $(CFLAGS) ***$^***

Explanation*:*

1. The **%** character serves as a wildcard to refer to all filenames ending with .o and .c.
2. Macro meaning are:
    a. **$@** Refers to the left side of the colon(:) or the target name.
    b. **$^** Refers to the right side of the colon(:) or the complete depedency list.
    c. **$<** Refers to the first dependency in the dependency list. Useful if you have a constant second dependency (like a header file).

REFERENCES:

1. http://www.gnu.org/software/make/manual/make.html
2. http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

PROCESS INFO:

In most Linux & Windows distributions with a gui, you have an Task Manager utility which lets you view what all processes are running in the system, and certain information about each process (the process id, memory, cpu, etc).

These programs have their root in a very old, and still widely used command-line process viewing utility called **top.**

This utility is generally shipped with most Linux distributions.

Using the top command:

The **top** will display a continually updating report of system resource usage.

- The top portion of the report lists information such as the system time, uptime, CPU usage, physical ans swap memory usage, and number of processes.
- Below that is a list of the processes sorted by CPU utilization.
- You can modify the output of top while is is running.
  - Hit an **i** to toggle idle processes.
  - Hitting **M** will sort by memory usage.
  - **S** will sort by how long they processes have been running
  - **P** will sort by CPU usage again.
- You can also modify processes from within the top command.
  - Use **u** to view processes owned by a specific user
  - **k** to kill processes
  - **r** to renice them (change the process' priority, etc)

Further Useful top shortcuts: http://www.thegeekstuff.com/2010/01/15-practical-unix-linux-top-command-examples/