XSEDE Scholars
Introduction to C

Profiling & Debugging Serial Programs

What is GDB?

The GNU Project debugger.

GDB allows you to see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed.

Why Use GDB?

GDB can help you track down and fix bugs in just a few minutes

Most of what you need to accomplish with GDB can be done with a small set of commands.

Why use GDB?

GDB can do four main kinds of things to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Compile For GDB

```
$ gcc -g src.c -o prog
```

To start a debugging session type:

```
$ gdb prog -q
```

[The `-q` option turns off licensing messages]

debugme.c – A Program To Debug

```
/*
 * debugme.c - poorly written program to debug
 */

#include <stdio.h>
#define BIGNUM 5000

void index_to_the_moon(int ary[]);

int main(int argc, char *argv[])
{
    int intary[100];
    index_to_the_moon(intary);
    return 0;
}

void index_to_the_moon(int ary[])
{
    int i;
    for (i = 0; i < BIGNUM; ++i)
    ary[i] = i;
}
```

Segmentation Fault

When you execute `./debugme` it will cause a segmentation fault.

```
lslogin2% gcc -g debugme.c -o debugme
lslogin2% ./debugme
Segmentation fault
```

A segmentation fault occurs anytime a program attempts to access memory that doesn't explicitly belong to it.

Run The Debugger

```
lslogin2% gdb debugme -q
(gdb) run
Starting program: /home/jlockman/debug/debugme

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400479 in index_to_the_moon
(ary=0x7fbfffe9e0) at debugme.c:21
21                  ary[i] = i;
```

Inspect the Code in the Debugger

```
(gdb) backtrace
#0  0x0000000000400479 in index_to_the_moon (ary=0x7fbfffe9e0)
at debugme.c:21
#1  0x0000000000400446 in main (argc=103, argv=0x6500000064)
at debugme.c:13
```

The `list` Command

`(gdb)` **`list 10,24`** - will display code lines 10-24

```
(gdb) list
13          index_to_the_moon(intary);
14          return 0;
15  }
16
17  void index_to_the_moon(int ary[])
18  {
19          int i;
20          for (i = 0; i < BIGNUM; ++i)
21                  ary[i] = i;
22  }
```

Examine The Data with `print`

```
(gdb) print ary@10
$2 = {0x7fbfffe9e0, 0x7fbfffeb80, 0x400446, 0x100000000,
0x300000002, 0x500000004, 0x700000006, 0x900000008,
0xb0000000a, 0xd0000000c}
```

```
(gdb) print ary[1]@5
$3 = {1, 2, 3, 4, 5}
```

Print five values stored in ary beginning with the first element.

Examine The Data with `print`

```
(gdb) print i
$1 = 1416
(gdb) print ary[i]
Cannot access memory at address 0x7fc0000000
(gdb)
```

The result of the command `print ary[i]` makes it clear that the program does not have access to the memory location specified .

The `whatis` Command

```
(gdb) whatis i
type = int
(gdb) whatis ary
type = int *
(gdb) whatis index_to_the_moon
type = void (int *)
```

debugme.c – The Bug

```
/*
 * debugme.c - poorly written program to debug
 */

#include <stdio.h>
#define BIGNUM 5000

void index_to_the_moon(int ary[]);

int main(int argc, char *argv[])
{
    int intary[100];
    index_to_the_moon(intary);
    return 0;
}

void index_to_the_moon(int ary[])
{
    int i;
    for (i = 0; i < BIGNUM; ++i)
    ary[i] = i;
```

Next Example: lucky.c

```
#include <stdio.h>
int ReturnSeven(){
    int num = 7;
    return num;
}
int ReturnThree(){
    int num = 3;
    return num;
}
int main(void){
    int seven;
    int three;
    seven = ReturnSeven();
    three = ReturnThree();
    printf("My lucky numbers are: =  %d ,%s\n",seven,three);
    return 0;
}
```

Another Segmentation Fault!

```
lslogin2% gcc lucky.c -o lucky
lslogin2% ./lucky
Segmentation fault
```

Why?

Compile for GDB

```
lslogin2% gcc -g lucky.c -o lucky
lslogin2% gdb lucky -q
(gdb) run
Starting program:
  /home/jlockman/debug/lucky


Program received signal SIGSEGV,
  Segmentation fault.
0x0000003000270a10 in strlen () from
  /lib64/tls/libc.so.6
(gdb)
```

Try A Different Method

This time GDB did not give us a line number to look for an error, let us try something different.

Set **`Breakpoints`**

Use **`step`**

Setting Breakpoints

It may be useful to stop execution at some point so that we can look at values of certain variables at the given point

You can set breakpoints at a line number:

```
(gdb) break linenum
```

Or when the code enters a function:

```
(gdb) break funcname
```

Set Breakpoints

```
(gdb) break main
Breakpoint 1 at 0x4004a0: file lucky.c,
  line 13.
(gdb) break ReturnSeven
Breakpoint 2 at 0x40047c: file lucky.c,
  line 3.
(gdb) break ReturnThree
Breakpoint 3 at 0x40048c: file lucky.c,
  line 7.
```

Step Through

```
(gdb) run
Starting program: /home/jlockman/debug/lucky

Breakpoint 1, main () at lucky.c:13
13      seven = ReturnSeven();
```

Step – to function ReturnSeven

```
(gdb) step

Breakpoint 2, ReturnSeven () at lucky.c:3
3       int num = 7;
(gdb) step
4       return num;
(gdb) step
5 }
```

Step – Takes Us Back to main()

```
(gdb) step
main () at lucky.c:14
14      three = ReturnThree();
```

Step – to function ReturnThree

```
(gdb) step

Breakpoint 3, ReturnThree () at lucky.c:7
7       int num = 3;
(gdb) step
8       return num;
(gdb) step
9 }
```

Step – Brings Us Back to main()

```
(gdb) step
main () at lucky.c:15
15      printf("My lucky numbers are: =  %d
  ,%s\n",seven,three);
(gdb) step
```

```
Program received signal SIGSEGV, Segmentation fault.
0x0000003000270a10 in strlen () from /lib64/tls/libc.so.6
```

The Bug

We see the segfault does not occur in the sub functions, it happens after the `printf` statement in `main()`.

After further inspection we can see our error:

```
15      printf("My lucky numbers are: =  %d,%s\n",seven,three);
```

Last Example – Using `watch`

```
#include <stdio.h>
int main(void){
    int count = 0;
    count += 4;
    count --;

    if(count == 0){ printf("count is zero!\n"); }
    else if(count == 1){ printf("count is One!\n"); }
    else if(count = 2){ printf("count is Two!\n"); }
    else if(count == 3){ printf("count is Three!\n"); }
    else if(count == 4){ printf("count is Four!\n"); }
    return 0;
}
```

Whats Wrong?

From our declaration of 'count' the final value should be 3.

```
int count = 0;      //count = 0
count += 4;         //count = 4
count --;           //count = 3
```

However when we run the program it always prints “Count is Two!”

```
lslogin2% ./watch
count is Two!
```

Lets 'Watch' the Value of count

```
lslogin2% gcc -g watch.c -o watch
lslogin2% gdb watch -q
(gdb) break main
Breakpoint 1 at 0x400480: file watch.c, line 3.
(gdb) run
Starting program: /home/00944/jlockman/debug/tmp/watch

Breakpoint 1, main () at watch.c:3
3             int count = 0;
(gdb) watch count
Hardware watchpoint 2: count
```

Continue Execution

```
(gdb) continue
Continuing.
Hardware watchpoint 2: count

Old value = 0
New value = 4
main () at watch.c:5
5       count --;
```

Continue Execution

```
(gdb) continue
Continuing.
Hardware watchpoint 2: count


Old value = 4
New value = 3
main () at watch.c:7
7       if(count == 0){ printf("count is zero!\n");
  }
```

Continue & Find The Bug

```
(gdb) continue
Continuing.
Hardware watchpoint 2: count


New value = 2
0x00000000004004c9 in main () at watch.c:9
9       else if(count = 2){ printf("count is Two!\n"); }
```

The Bug, A Missing '='

```
#include <stdio.h>
int main(void){
    int count = 0;
    count += 4;
    count --;

    if(count == 0){ printf("count is zero!\n"); }
    else if(count == 1){ printf("count is One!\n"); }
    else if(count = 2){ printf("count is Two!\n"); }
    else if(count == 3){ printf("count is Three!\n"); }
    else if(count == 4){ printf("count is Four!\n"); }
    return 0;
}
```

Profiling Your Code with `gprof`

Profiling allows you to see where your code spends the bulk of it's time

This information can show you which pieces of your program are slower than you expected

It can also help with programs that are too large or too complex to analyze by reading the source

Example: taylor.c

In taylor.c we will compute e*pi using Taylor expansion to calculate e and pi

We have a function for calculating e and pi, and a main function.

We will use `gprof` to profile this program and find out which sections of code are using the most cycles.

3 Steps to Profiling

You must compile and link your program with profiling enabled

You must execute your program to generate a profile data file

You must run `gprof` to analyze the profile data

Compile & Link with Profiling Enabled

Use the -pg option with gcc when compiling

```
lslogin2% gcc -pg taylor.c -o taylor
```

Execute the Program

```
lslogin2% ./taylor
e*pi by Taylor Expansion =  8.539734
```

Execution creates a file named `gmon.out`

Run `gprof`

```
lslogin2% gprof taylor gmon.out
```

Or you can redirect the output to a file.

```
lslogin2$ gprof taylor gmon.out > mylog.txt
```

Analyze Your Data – The Flat Profile

The flat profile shows the total amount of time your program spent executing each function

Flat profile:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 98.74  3.81      3.81         1     3.81     3.81    CalcPi
  2.09  3.89      0.08         1     0.08     0.08    CalcE
```

Analyze Your Data – The 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.

Analyze Your Data – The Call Graph

```
     Call graph (explanation follows)
granularity: each sample hit covers 2 byte(s) for 0.26% of 3.89 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0       0.00    3.89                 main [1]
                   3.81    0.00   1/1               CalcPi [2]
                   0.08    0.00   1/1               CalcE [3]
-----------------------------------------------
                   3.81    0.00   1/1                main [1]
[2]     97.9       3.81    0.00   1             CalcPi [2]
-----------------------------------------------
                   0.08    0.00   1/1               main [1]
[3]      2.1       0.08    0.00   1             CalcE [3]
-----------------------------------------------
```

Manuals and References

GDB manual page `man gdb`

“Linux Bible” by Chris Negus

“Practical C++ Programming – Nutshell Handbook” by Steve Oualline

“Running Linux” by Matt Welsh, Lars Kaufman