CMPSC 311 - Introduction to Systems Programming
Module: Debugging

Professor Patrick McDaniel
Fall 2014
Debugging

• Often the most complicated and time-consuming part of developing a program is *debugging*.
  ‣ Figuring out where your program diverges from your idea of what the code should be doing.
  ‣ Confirm that your program is doing what you expect to be doing.
  ‣ Finding and fixing bugs ...
Printing/Logging

- One way to debug is to print out the values of variables and memory at different points
  - e.g., `printf("My variable value is %d", myvar);`
  - Logging (such as `LogMessage()`) provides more sophisticated interfaces to simple prints, log to file
- Turning on an off “debug levels”
  - `LOG_INFO_LEVEL`
  - `LOG_WARNING_LEVEL`
  - `LOG_ERROR_LEVEL`
  - `LOG_OUTPUT_LEVEL`

```c
enableLogLevels( LOG_INFO_LEVEL );
...
logMessage( LOG_OUTPUT_LEVEL, "The log message is %d", value );
...
Fri Oct 18 10:26:04 2013 [OUTPUT] The log message is 11
```
Assert

• `assert()` is a function provided by C that allows you to place statements in code that must always be true, where the process SEGFAULTs if it is not
  ‣ This is a great tool for checking to make sure your assumptions about inputs/logic are always true
  ‣ Syntax:

    ```c
    assert( expression );
    ```

```c
#include <assert.h>

int factorial( int i ) {
    assert( i>=0 );  // ** CHECK **
    if ( i <= 1 ) {
        return( i );
    }
    return( factorial(i-1)*i );
}
```

$ ./debugging
Factorial : 5! = 120
debugging: debugging.c:6: factorial: 
Assertion `i>=0' failed.
Aborted (core dumped)
$
The debugger

• A debugger is a program that runs your program within a controlled environment:
  ‣ Control aspects of the environment that your program will run in.
  ‣ Start your program, or connect up to an already-started process.
  ‣ Make your program stop for inspection or under specified conditions.
  ‣ Step through your program one line at a time, or one machine instruction at a time.
  ‣ Inspect the state of your program once it has stopped.
  ‣ Change the state of your program and then allow it to resume execution.

• In UNIX/Linux environments, the debugger used most often is **gdb** (the GNU Debugger)
gdb

- You run the debugger by passing the program to `gdb`

  
  $ gdb [program name]

- This is an interactive terminal-based debugger

- Invoking the debugger does not start the program, but simply...

You can always get help for any command in `gdb` by typing `help [command]`
Running the program

• Once you enter the program, you must start the program running, using the `run` command

```gdb
(run) run
Starting program: /home/mcdaniel/src/debugging/debugging
Arguments (1), last arg [/home/mcdaniel/src/debugging/debugging]
Factorial : 5! = 120
[Inferior 1 (process 36524) exited normally]
(gdb)
```

• If you have arguments to pass to the program, simply add them to the `run` command line

```gdb
(run) run sample
Starting program: /home/mcdaniel/src/debugging/debugging sample
warning: no loadable sections found in added symbol-file system-supplied DSO at 0x7ffff7ffa000
Arguments (2), last arg [sample]
Factorial : 5! = 120
[Inferior 1 (process 36538) exited normally]
(gdb)
```
Looking at code

- While in the debugger you often want to look at regions of code, so use the `list` command
  - shows 10 lines at a time
  - you can specify a line number (in the current file),
  - or specify a function name

```
(gdb) list 4
1       #include <stdio.h>
2       #include <assert.h>
3
4       int factorial( int i ) {
5           assert( i>=0 );
6           if ( i == 1 ) {
7               return( 1 );
8           }
9           return( factorial(i-1)*i );
10      }
11
12 int main( int argc, char *argv[] ) {
13    return( factorial(i-1)*i );
14 }
15
16 int main( int argc, char *argv[] ) {
17    printf( "Arguments (%d), last arg [%s]
\n", argc, argv[argc-1] );
18    }
19
(gdb) l main
8       return( 1 );
9    }
10         return( factorial(i-1)*i );
11       }
12
13 int main( int argc, char *argv[] ) {
14 15         if ( argc > 0 ) {
16             printf( "Arguments (%d), last arg [%s] \\
17             argc, argv[argc-1] );
18         }
19
(gdb)
```

- Most commands are aliased with single character (1)
Breakpoints

- A **breakpoint** is a position in the code you wish for the debugger to stop and wait for your commands
  - Breakpoints are set using the break (b) command
    ```plaintext
    break [function_name | line_number]
    ```
  - Each one is assigned a number you can reference later

- You can delete the breakpoint by using the delete (d) command
  ```plaintext
  delete [breakpoint_number]
  ```

```plaintext
(gdb) b factorial
Breakpoint 1 at 0x400587: file debugging.c, line 6.
(gdb) b 16
Breakpoint 2 at 0x4005db: file debugging.c, line 16.
(gdb) delete 1
(gdb) d 2
```
Conditional Breakpoints

• A conditional *breakpoint* is a point where you want the debugger only if the condition holds
  ‣ Breakpoints are set using the `cond` command

  \[
  \text{cond [breakpoint_number] (expr)}
  \]

  ```
  (gdb) 1 6
  6       assert( i>=0 );
  (gdb) b 6
  Breakpoint 1 at 0x400587: file debugging.c, line 6.
  (gdb) cond 1 i<=1
  (gdb) r
  Starting program: /home/mcdaniel/src/debugging/debugging
  warning: no loadable sections found in added symbol-file system-supplied DSO at
  0x7ffff7ffa000
  Argumentts (1), last arg [/home/mcdaniel/src/debugging/debugging]
  Breakpoint 1, factorial (i=1) at debugging.c:6
  6       assert( i>=0 );
  (gdb) c
  Continuing.
  Z = 24
  [Inferior 1 (process 37293) exited normally]
  (gdb)
  ```
Conditional Breakpoints

• A conditional **breakpoint** is a point where you want the debugger only if the condition holds
  ‣ Alternately, breakpoints can be set with **if** expression

    \[\text{b [line | function] if (expr)}\]

```
(gdb) l 27
22    return( 0 ); // breakpoint here
23 }
24
25    int factorial( int i ) {
26
27        assert( i>=0 );
28        if ( i == 1 ) {
29            return( 1 );

(gdb) b 27 if i<=1
Breakpoint 1 at 0x400767: file debugging.c, line 27.

(gdb) r
Starting program: /home/mcdaniel/scourses/cmpsc311-f14/src/debugging/debugging

Breakpoint 1, factorial (i=1) at debugging.c:27
27        assert( i>=0 );
```
Seeing breakpoints

• If you want to see your breakpoints use the `info breakpoints` command

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400587 in factorial at debugging.c:6
2       breakpoint     keep y   0x00000000004005f3 in main at debugging.c:16
(gdb)
```

• The info command allows you see lots of information about the state of your environment and program

```
(gdb) help info
Generic command for showing things about the program being debugged.

List of info subcommands:

  info address -- Describe where symbol SYM is stored
  info all-registers -- List of all registers and their contents
  info args -- Argument variables of current stack frame
  ...
```

Examining the stack

- You can always tell where you are in the program by using the `where` command, which gives you a stack and the specific line number you are one.

```plaintext
(gdb) where
#0  factorial (i=1) at debugging.c:6
#1  0x000000000004005c0 in factorial (i=2) at debugging.c:10
#2  0x000000000004005c0 in factorial (i=3) at debugging.c:10
#3  0x000000000004005c0 in factorial (i=4) at debugging.c:10
#4  0x0000000000040063b in main (argc=1, argv=0x7fffffffe4f8) at debugging.c:21
```
Climbing and descending the stack

- You can move up and down the stack and see variables by using the **up** and **down** commands

```c
int factorial( int i ) {
    assert( i>=0 );
    if ( i == 1 ) {
        return( 1 ); // Breakpoint here
    }
    return( factorial(i-1)*i );
}
```

Breakpoint 1, factorial (i=1) at debugging.c:29
29    return( 1 );
(gdb) p i
$1 = 1
(gdb) up
#1 0x0000000000040079f in factorial (i=2) at debugging.c:31
31    return( factorial(i-1)*i );
(gdb) p i
$2 = 2
(gdb) up
#2 0x0000000000040079f in factorial (i=3) at debugging.c:31
31    return( factorial(i-1)*i );
(gdb) p i
$3 = 3
(gdb) down
#1 0x0000000000040079f in factorial (i=2) at debugging.c:31
31    return( factorial(i-1)*i );
(gdb) p i
$4 = 2
(gdb) down
#0  factorial (i=1) at debugging.c:29
29    return( 1 );
(gdb) p i
$5 = 1
(gdb)
Printing variables

- At any point in the debug session can print the value of any variable you want by printing its value using
  ```
  print [/<format>] variable
  ```
- You can modify the output formatted with o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), and s(string)

```c
int myvalues( int x, int y, int z ) {
  char values[] = { 0x1, 0x2, 0x3, 0x4 };  
  uint32_t val1 = 0xff5566ff;
  float val2 = 2.45678;

  val2 = (float) val1;
  memset( values, 0xff, 4 );
  return( 0 ); // breakpoint here
}
```
Examining memory

- You examine memory regions using the `x` command:
  \[ x [/<\text{num}>/<\text{format}>/<\text{size}>] \text{ address} \]
- You can modify the output using a number of values formatted with `[oxdutfais]` type and size are `b`(byte), `h`(halfword), `w`(word), `g`(giant, 8 bytes).

```c
int myexamine( int x ) {
    char *buf = NULL;
    uint32_t val = 0;
    buf = malloc( 8 );
    memset( buf, 0xef, 8 );
    return( 0 ); // breakpoint here
}
```

```
(gdb) x &x
0x7fffffffe3cc: 0x00000004
(gdb) x buf
0x602010: 0xefefefef
(gdb) x /t buf
0x602010: 1110111111011111101111101111
(gdb) x /8xb &buf
0x7fffffffe3d8: 0x10 0x20 0x60 0x00 0x00 0x00 0x00 0x00
(gdb) x /8xb &buf
0x7fffffffe3d8: 0x10 0x20 0x60 0x00 0x00 0x00 0x00 0x00
(gdb) p buf
$2 = 0x602010 "\357\357\357\357\357\357\357\357\357\357", <incomplete sequence \357>
```
Walking the program

• There are four ways to advance the program in gdb
  ▸ `next (n)` steps the program forward one statement, regardless of the kind of statement it is on

```c
int factorial( int i ) {
    if ( i == 1 ) {
        return( 1 );
    }
    return( factorial(i-1)*i );
}

int main( int argc, char *argv[] ) {
    int x = factorial(5);
    printf( "Factorial : %d! = %d\n", 5, x );
    return( 0 );
}
```
Walking the program

- There are four ways to advance the program in gdb
  - `next (n)` steps the program forward one statement, regardless of the kind of statement it is on
  - `step (s)` moves the program forward one statement, but “steps into” a program-defined function
Walking the program

- There are four ways to advance the program in gdb
  - `next (n)` steps the program forward one statement, regardless of the kind of statement it is on
  - `step (s)` moves the program forward one statement, but “steps into” a program-defined function
  - `continue (c)` continues running the program from that point till it terminates or hits another breakpoint

```c
int main( int argc, char *argv[] ) {
    int x = factorial(5);
    printf( "Factorial : %d! = %d\n", 5, x );
    return( 0 );
}
```
Walking the program

- There are four ways to advance the program in gdb
  - next (n), step (s), continue (c), ... and
  - finish (fin) continues until the function returns

```c
int factorial( int i ) {
    if ( i == 1 ) {
        return( 1 );
    }
    return( factorial(i-1)*i );
}

int main( int argc, char *argv[] ) {
    int x = factorial(5);
    printf( "Factorial : %d! = %d\n", 5, x );
    return( 0 );
}
```
#include <stdio.h>  
#include <assert.h>  

int factorial( int i ) {  

    assert( i>=0 ); // Breakpoint here  
    if ( i == 1 ) {  
        return( 1 );  
    }  
    return( factorial(i-1)*i );  
}

int main( int argc, char *argv[] ) {  

    if ( argc > 0 ) {  
        printf( "Arguments (%d), last arg [%s]\n",  
                argc, argv[argc-1] ); // Breakpoint here  
    }  
    printf( "Factorial : %d! = %d\n", 5, factorial(5) );  
    // factorial( -1 );  
    return( 0 );

(gdb) b factorial  
Breakpoint 1 at 0x400587: file debugging.c, line 6.  
(gdb) b 17  
Breakpoint 2 at 0x4005db: file debugging.c, line 17.  
(gdb)
Putting it all together

(gdb) r
Starting program: /home/mcdaniel/src/debugging/debugging
Breakpoint 2, main (argc=1, argv=0x7fffffffe4f8) at debugging.c:17
    argc, argv[argc-1] );
(gdb) n
16  printf( "Argumentts (%d), last arg [%s]\n",
(gdb) n
Argumentts (1), last arg [/home/mcdaniel/src/debugging/debugging]
19  printf( "Factorial : %d! = %d\n", 3, factorial(3) );
(gdb) s
Breakpoint 1, factorial (i=3) at debugging.c:6
6   assert( i>=0 );
(gdb) c
Continuing.

Breakpoint 1, factorial (i=2) at debugging.c:6
6   assert( i>=0 );
(gdb) c
Continuing.

Breakpoint 1, factorial (i=1) at debugging.c:6
6   assert( i>=0 );
(gdb) c
Continuing.
Factorial : 3! = 6
[Inferior 1 (process 37115) exited normally]
(gdb)

int factorial( int i ) {
    assert( i>=0 ); // Breakpoint here
    if ( i == 1 ) {
        return( 1 );
    }
    return( factorial(i-1)*i );
}

int main( int argc, char *argv[] ) {
    if ( argc > 0 ) {
        printf( "Argumentts (%d), last arg [%s]\n",
            argc, argv[argc-1] ); // Breakpoint here
        printf( "Factorial : %d! = %d\n", 3, factorial(3) );
        return( 0 );
    }
Watchpoints

- *Watchpoints* (also known as a *data breakpoint*) stop execution whenever the value of an variables changes, *without having to predict a particular place where this may happen.*
  - The simplest form is simply waiting for a variable to change

```c
int z = 0;
...
z = factorial(4);
printf("Z = %d\n", z);
```

(gdb) `watch z`
Hardware watchpoint 6: z
(gdb) c
Continuing.
Arguments (1), last arg [/home/mcdaniel/siis/courses/cmpsc311-f13/slides/src/debugging/
debugging]
Factorial : 3! = 6
Hardware watchpoint 6: z

Old value = 0
New value = 24
main (argc=1, argv=0x7fffffffe4f8) at debugging.c:22
22 printf("Z = %d\n", z );
(gdb)
```
For next time

• Download and compile the program listed on the course website (as of Thursday afternoon), bring in your laptop if you can.