codeplay

Why should I use LLDB?

Deepak Panickal
Ewan Crawford

- Hiya!

- We’re debugger engineers at Codeplay
- Working on LLDB for the past three years on customer projects
- In the past year, have been adapting LLDB for Qualcomm’s Hexagon DSP
- Upstreaming relevant patches to the community

- Codeplay!
- Heterogeneous compiler experts
- 35 talented engineers
- Based out of Edinburgh, Scotland

Edinburgh Castle

- We work on
  - R&D (both self and externally funded)
  - Work for hire contracting, compiler/debugger tech
  - Standards via bodies such as Khronos

codeplay

- Motivation
- LLDB Command-line and MI Interface
- Leveraging LLVM Libraries
- LLDB C++ and Python API
- Future Developments
- Summary
- Q/A

Motivation

- LLDB has a clean, **maintainable**, plugin architecture
- Works on all major platforms (Windows support coming up to speed)
- Default debugger in Xcode on Mac OS X
- Inviting more people to start using it as their primary debugger
- We aim to showcase LLDB's features from the **user** point of view

Command-line Interface

- LLDB's command-line interface is **verbose**
- Commands have **short** forms too
- **Tab completion** of commands (not on Windows yet)
- Detailed settings
- Easy jump from **GDB to LLDB** as a lot of commands are similar
- http://lldb.llvm.org/lldb-gdb.html

(gdb) run (gdb) r	(lldb) process launch (lldb) run (lldb) r
(gdb) b main	(lldb) breakpoint set --name main (lldb) br s -n main (lldb) b main
(gdb) x/4xw 0xbffff3c0	(lldb) memory read --size 4 --format x --count 4 0xbffff3c0 (lldb) me r -s4 -fx -c4 0xbffff3c0 (lldb) x -s4 -fx -c4 0xbffff3c0 (lldb) x/4xw 0xbffff3c0
(gdb) bt	(lldb) thread backtrace (lldb) bt

codeplay

- **help** command displays all supported commands

```
(lldb) help
Debugger commands:

  apropos          -- Find a list of debugger commands related to a particular word/subject.
  breakpoint       -- A set of commands for operating on breakpoints. Also see _regexp-break.
  expression       -- Evaluate a C/ObjC/C++ expression in the current program context, using user defined
                      variables and variables currently in scope.
  frame            -- A set of commands for operating on the current thread's frames.
    . . .
  print            -- ('expression --')  Evaluate a C/ObjC/C++ expression in the current program context,
                       using user defined variables and variables currently in scope.
  q                -- ('quit')  Quit out of the LLDB debugger.
  r                -- ('process launch -c /bin/sh --')  Launch the executable in the debugger.
  s                -- ('thread step-in')  Source level single step in specified thread (current thread,
                       if none specified).
  step             -- ('thread step-in')  Source level single step in specified thread (current thread,
                       if none specified).
  t                -- ('thread select')  Select a thread as the currently active thread.
  x                -- ('memory read')  Read from the memory of the process being debugged.

For more information on any command, type 'help <command-name>'.
```

codeplay

- help <command-name>

```
(lldb) help breakpoint
The following subcommands are supported:
  clear   -- Clears a breakpoint or set of breakpoints in the executable.
  delete  -- Delete the specified breakpoint(s).  If no breakpoints are specified, delete them all.
  enable  -- Enable the specified disabled breakpoint(s). If no breakpoints are specified, enable all of
them.
  list-- List some or all breakpoints at configurable levels of detail.
  set -- Sets a breakpoint or set of breakpoints in the executable.
```

- help <command-name> <subcommand-name>

```
(lldb) help breakpoint set
      Sets a breakpoint or set of breakpoints in the executable.
Syntax: breakpoint set <cmd-options>

  -c <expr> ( --condition <expr> )
           The breakpoint stops only if this condition expression evaluates to true.

  -f <filename> ( --file <filename> )
```

- Any unique **short** form of a command can be used

```
(lldb) watchpoint set variable count

(lldb) w s v count
```

- **Settings** are detailed and easily accessible

```
(lldb) settings set target.process.stop-on-sharedlibrary-events on

(lldb) settings set target.output-path stdout.txt
```

codeplay

- **apropos** for searching commands related to a word

```
(lldb) apropos disassem

The following built-in commands may relate to 'disassem':
  disassemble -- Disassemble bytes in the current function, or elsewhere in the executable program as specified by the
user.


The following settings variables may relate to 'disassem':

  disassembly-format -- The default disassembly format string to use when disassembling instruction sequences.
  stop-disassembly-count -- The number of disassembly lines to show when displaying a stopped context.
  stop-disassembly-display -- Control when to display disassembly when displaying a stopped context.
  target.x86-disassembly-flavor -- The default disassembly flavor to use for x86 or x86-64 targets.
  target.use-hex-immediates -- Show immediates in disassembly as hexadecimal.
  target.hex-immediate-style -- Which style to use for printing hexadecimal disassembly values.
```

codeplay

- **Rich** debugging with detailed process state information

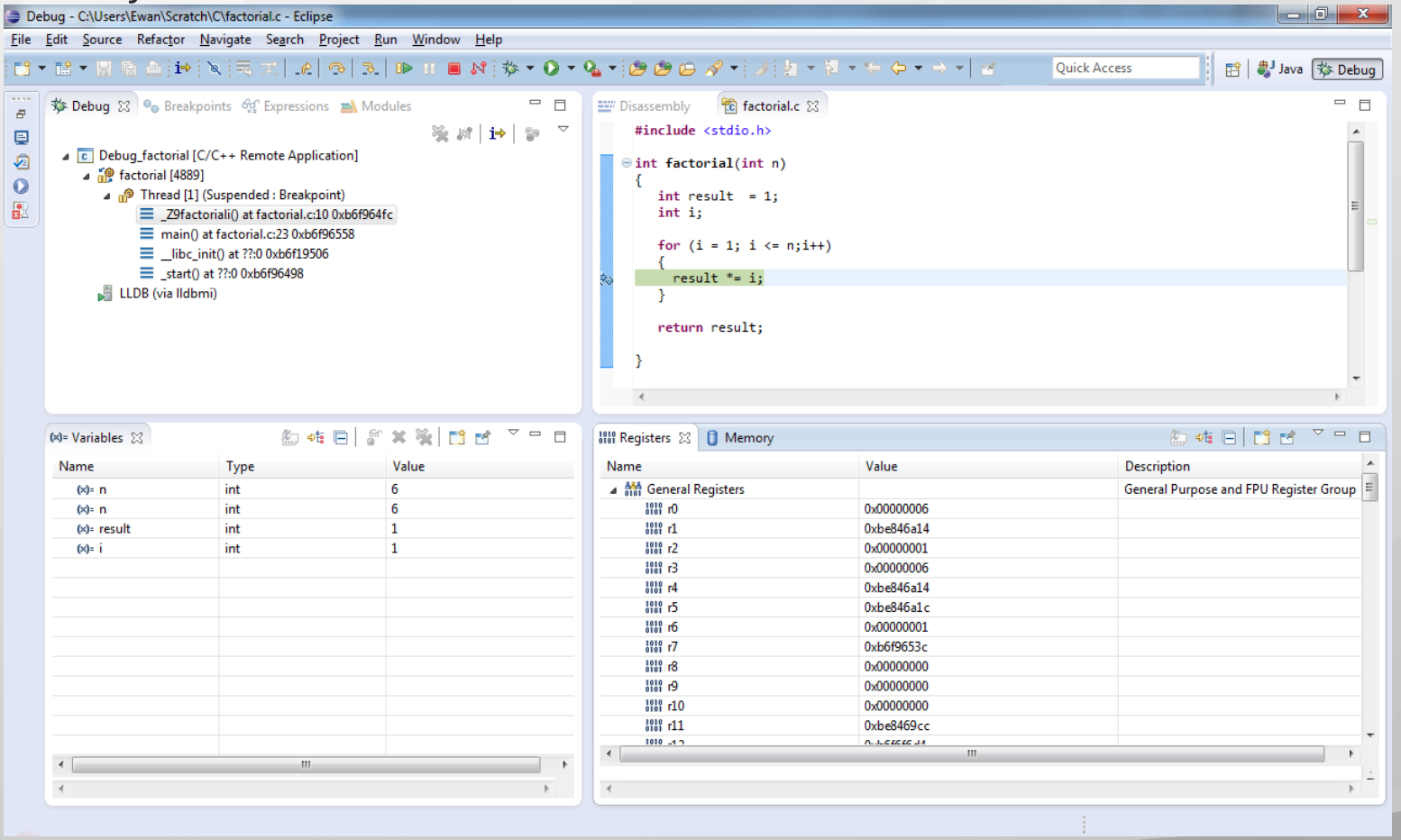
```
(lldb) file a.out
Current executable set to 'a.out' (x86_64).

(lldb) breakpoint set --name main --file example.c
Breakpoint 1: where = a.out`main + 8 at example.c:22, address = 0x00000000004005d7

(lldb) b factorial
Breakpoint 2: where = a.out`factorial + 7 at example.c:5, address = 0x00000000004005a4

(lldb) r
Process 2210 launched: '/home/ewan/Desktop/Scratch/talk/c_example/a.out' (x86_64)
Process 2210 stopped
* thread #1: tid = 2210, 0x00000000004005d7 a.out`main + 8 at example.c:22, name = 'a.out', stop reason = breakpoint 1.1
      frame #0: 0x00000000004005d7 a.out`main + 8 at example.c:22
   19   {
   20
   21         int number;
-> 22         printf("Enter a number to calculate factorial of: ");
   23         scanf("%d",&number);
   24
   25         int fact = factorial(number);
(lldb)
```

LLDB-MI Interface

- Machine Interface(MI) is a **protocol** allowing a front end to communicate with the debugger in text form
- Many existing IDEs currently use GDB for their debugging through **GDB-MI**
- LLDB-MI understands the GDB-MI protocol and runs as a separate executable using the LLDB C++ API
- More details about the design of LLDB-MI
  http://www.codeplay.com/portal/lldb-mi-driver---part-1-introduction

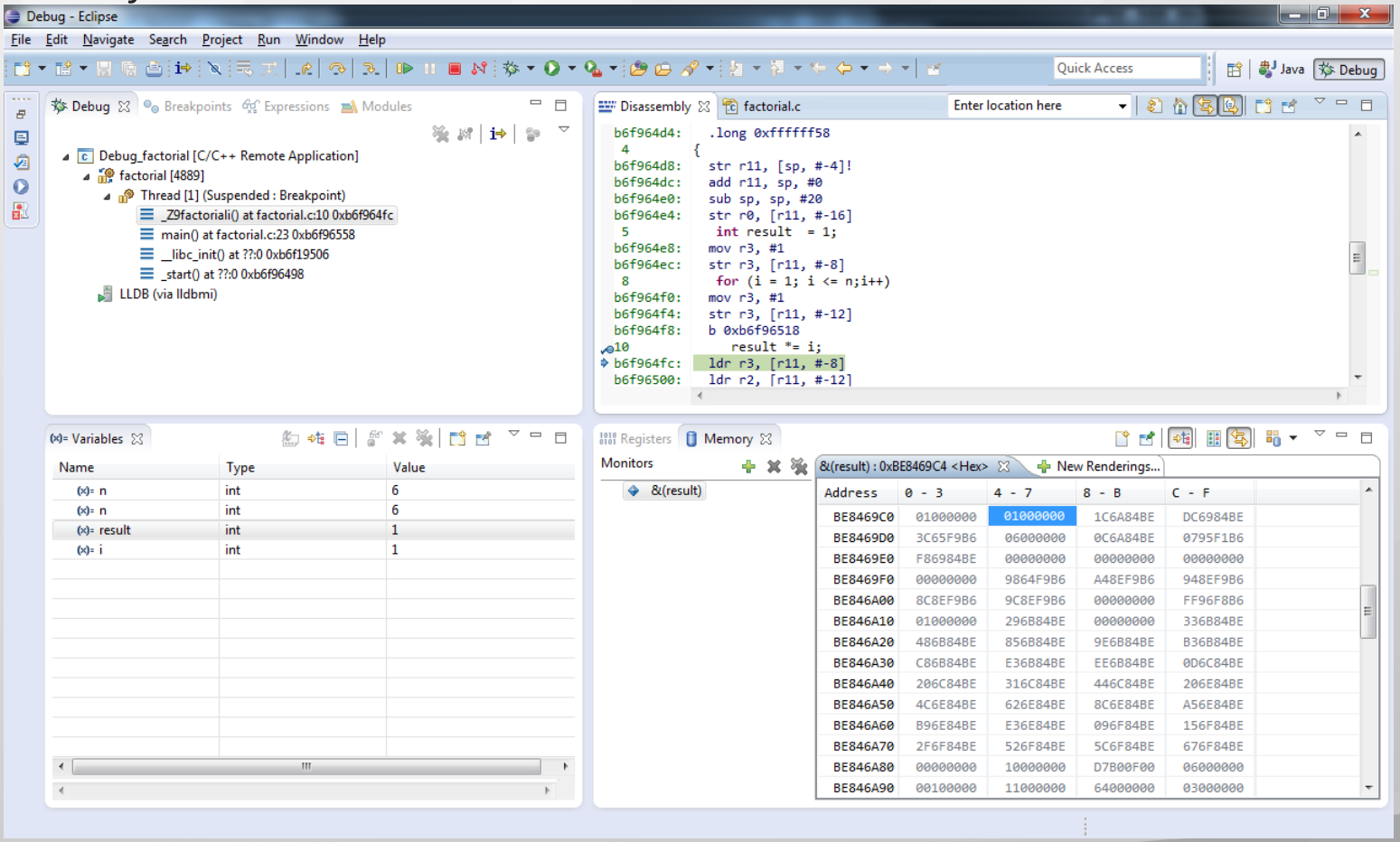
LLDB-MI Interface

codeplay



LLDB-MI Interface

codeplay



Leveraging LLVM Libraries

- Get all of the following for free
- Up to date **language support** with Clang compiler infrastructure
- Disassembling instructions for many architectures.
- LLDB has both IR interpreter for simple expressions and JIT-ting for complex **multi-line expressions** on supported architectures leveraging the power of Clang and LLVM

- **expression** command for expression evaluation

```
(lldb) help expr

      Evaluate a C/ObjC/C++ expression in the current program context, using user defined variables and
variables currently in scope. This command takes 'raw' input (no need to quote stuff).

Syntax: expression <cmd-options> -- <expr>

      -D <count> ( --depth <count> )
            Set the max recurse depth when dumping aggregate types (default is infinity).

      -F ( --flat )
            Display results in a flat format that uses expression paths for each variable or member.
....

Examples:

   expr my_struct->a = my_array[3]
   expr char c[] = "foo"; c[0]
```

codeplay

- **Multi-line** expression evaluation

```
(lldb) expr
Enter expressions, then terminate with an empty line to evaluate:
  1: int i = 0;
  2: for (;i<10;++i){
  3:   printf("%d\n",factorial(i));
  4: }
1
1
2
6
24
120
720
5040
40320
362880
```

- Can declare **local** variables

- **C++** expressions in a C program

```
(lldb) expr

Enter expressions, then terminate with an empty line to evaluate:
  1: auto square_lambda = [] (int i) { return (i*i);};
  2: int $squared = square_lambda(16);

(lldb) print $squared
(int) $squared = 256
```

- Variable types are deduced

```
(lldb) expr -T -- structVar
(complexStruct) $4 = {
  (unsigned int) firstInt = 2
  (long) secondInt = -1
  (char [3]) firstString = "abc"
  (char *) secondString = 0x00000000004005f4 "abc"
}
```

codeplay

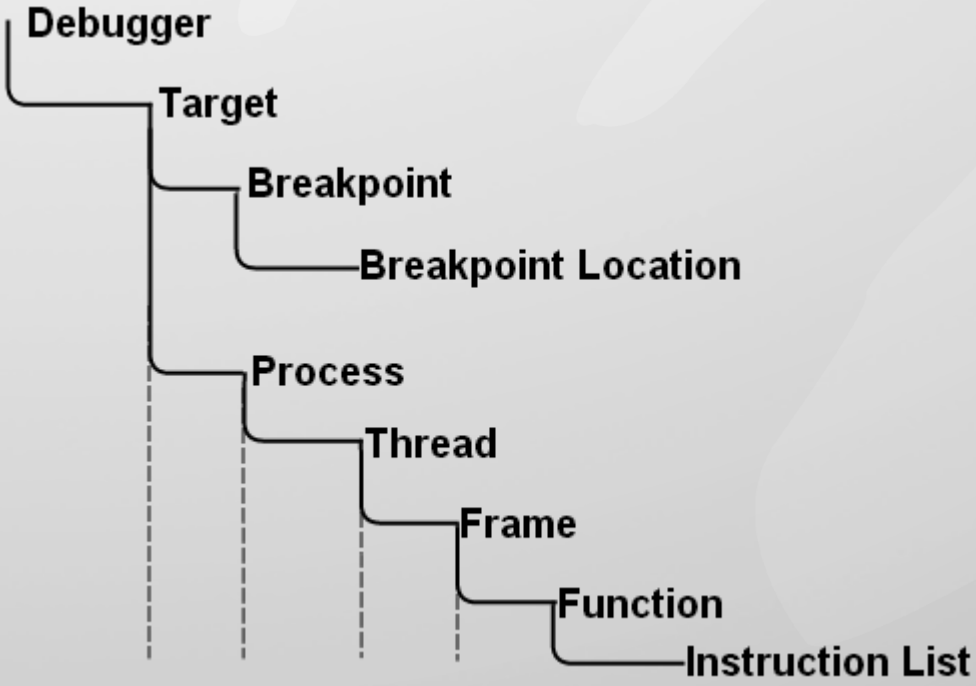
- Using the disassembler

```
(lldb) dis -n square(int) -m

a.out`square(int) at main.c:5
   4    int square(int n)
   5    {
   6
a.out`square(int):
      0x40052d <+0>:  pushq  %rbp
      0x40052e <+1>:  movq   %rsp, %rbp
      0x400531 <+4>:  movl   %edi, -0x4(%rbp)
a.out`square(int) + 7 at main.c:7
   6
-> 7      return n * n;
   8    }
->  0x400534 <+7>:  int3
      0x400535 <+8>:  cld
      0x400537 <+10>: imull  -0x4(%rbp), %eax
a.out`square(int) + 14 at main.c:8
   7      return n * n;
   8    }
   9
```

LLDB C++ API

- LLDB provides a **public C++ API** which allows features like disassembly and object file inspection to be integrated into user created C++ applications
- The API consists of a light **wrapper** around internal LLDB objects. e.g. Target and Frame
- API can be used through the LLDB shared library
- C++ API documentation http://lldb.llvm.org/cpp_reference/html/index.html

- API Object Organisation



LLDB Python API

- The LLDB API is available through Python script bindings
- Allows any Python script loading the LLDB module to **automate** repetitive debugging tasks
- API can be used inside LLDB's **embedded python** interpreter
- Python API documentation
  http://lldb.llvm.org/python_reference/index.html

- Python bindings are created using SWIG
  - SWIG (Simplified Wrapper and Interface Generator) provides a **wrapper** around C/C++ code, allowing it to be called by a different programming language
  - **Interface files** are used by SWIG to define how functions are made visible in the target language
  - SWIG is run at build time to create a **lldb.py** module
  - Developers can define SWIG interface files for **any supported language** they want to use LLDB functionality in. e.g. Java, C#, Lua

codeplay

```
import lldb

# Create a new debugger instance
debugger = lldb.SBDebugger.Create()
debugger.SetAsync (False)
target = debugger.CreateTargetWithFileAndArch ("./a.out", lldb.LLDB_ARCH_DEFAULT)

# Set breakpoint on function defined by command line argument. WARNING: No error checking.
main_bp = target.BreakpointCreateByName (sys.argv[1], target.GetExecutable().GetFilename());

process = target.LaunchSimple (None, None, os.getcwd()) # Launch process
if process.GetState() == lldb.eStateStopped:
    thread = process.GetThreadAtIndex (0)               # Get the first thread
    frame = thread.GetFrameAtIndex (0)                  # Get the first frame
    allVars = frame.get_all_variables()
    print("all variables: ")
    for var in allVars:                                 # Print all variables
          print str(var)
```

```
$ python printVars.py factorial
all variables:
(int) n = 6
(int) result = 0
(int) i = 0
```

- Using the embedded Python interpreter
  - Full featured interpreter
  - Can import external modules
  - (lldb) script ‘command’, will run without dropping into interpreter
  - (lldb) command script import ‘script file’, loads external scripts

```
(lldb) help script
     Pass an expression to the script interpreter for evaluation and return the results. Drop into the
interactive interpreter if no expression is given.  This command takes 'raw' input (no need to quote stuff).

Syntax: script [<script-expression-for-evaluation>]

(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()' or Ctrl-D.

>>> import math

>>> math.ceil(6.7)
7.0
```

codeplay

- The API can be used inside the interpreter through convenience variables
  - lldb.debugger
  - lldb.frame
  - lldb.target
  - lldb.process
  - lldb.thread

```
(lldb) script
>>> print lldb.frame
frame #0: 0x00000000004004f1 a.out`main + 4 at loop.c:4

>>> print lldb.frame.GetSP()
140737488346352

>>> print hex(lldb.frame.GetSP())
0x7fffffffdcf0L
```

codeplay

- New LLDB commands can also be created using Python functions
  - Work like native lldb commands
  - Clean way to extend LLDB functionality
  - Example from LLDB Python Reference
    http://lldb.llvm.org/python-reference.html

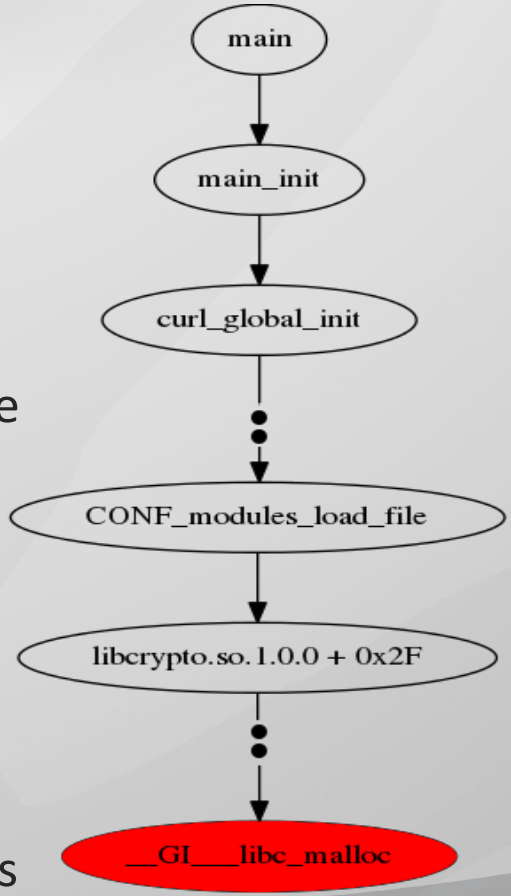
```
#~/ls.py
def ls(debugger, command, result, internal_dict):
    print >>result, (commands.getoutput('/bin/ls %s' % command))
```

```
(lldb) command script import ~/ls.py

(lldb) ls -l ~/LLVM/llvm/tools/lldb
total 88
drwxrwxr-x  4 ewan ewan 4096 Mar 30 16:32 cmake
-rw-rw-r--  1 ewan ewan 1205 Mar 30 16:32 CMakeLists.txt
```

codeplay

- One of the most powerful features the Python API enables is **running scripts** when breakpoints are hit
  - Python function definition:
    `breakpoint_function_wrapper(frame, bp_loc, dict)`
  - Function returns False to prevent LLDB from stopping when breakpoint is hit
  - Not stopping allows **profiling data** to be silently gathered every time a breakpoint is hit

- Breakpoint Script Example
  - **Walks up the call stack** on breakpoint hit and adds functions to call graph
  - Graph printed as image using pydot module
  - Graph generated from LLDB debugging the popular **curl** application
  - Script was attached to a conditional breakpoint on all calls to **malloc()** of 3 bytes



codeplay

```
# Full code available from https://github.com/EwanC/WhyShouldIUseLLDB
callGraph = CallGraph();                    # User defined class
root = callGraph.addNode("Root",-1);

def bpStack (frame, bp_loc, internal_dict): # Run when breakpoint is hit

       thread = frame.GetThread()
       numFrames = thread.GetNumFrames()

       lastnode = root                          # Parent function
       for f in reversed(range(0, numFrames)): # Walk the stack
           name = thread.GetFrameAtIndex(f).GetFunctionName()

           # Debug info not available
           if name == "???" or name == None:
               # Use location in module for name
               # Omitted here for brevity

           # Update call graph with function
           node = callGraph.update(name,lastnode,f)
           lastnode = node

       return False # LLDB doesn't stop when breakpoint is hit

def draw(): # Print graph to png image
       callGraph.graph.write_png('BPCallStack.png')
```

Future Developments

codeplay

- Windows support including improvement of **native Windows debugging** and better command-line features
- Improved LLDB-MI support for more IDEs
- Support for more **architectures** to the many LLDB already supports(X86, ARM, X86_64, ARM64, ...)
- **C# API** interface
- Lots of devs working to make it awesome!

Summary

- Why use LLDB?
  - **Reuse** LLVM components, ideal as part of an LLVM based toolchain
  - Supported on all major **platforms** and supports many architectures
  - **Embedded Python** interpreter with API
  - Powerful **multi-line** expression evaluation
  - Info on building LLDB is available at http://lldb.llvm.org/build.html

- Contact
  - Deepak Panickal, deepak@codeplay.com
  - Ewan Crawford, ewan@codeplay.com
  - Python examples, https://github.com/EwanC/WhyShouldIUseLLDB
  - For LLDB dev issues, the mailing list is lldb-dev@cs.uiuc.edu