LLDB Python Interface

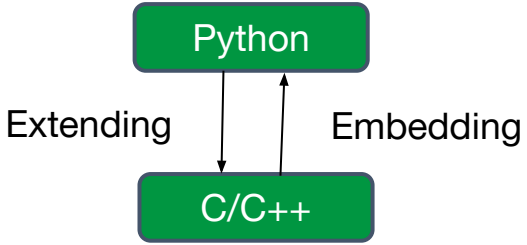
Omair Javaid

Introduction

- Interfacing Python with C/C++ applications
- LLDB architecture and role of Python
- LLDB interface with Python
- Using Python from LLDB command-line interface - Examples
- Using LLDB from Python - Example

Extending or Embedding Python

- **Extending - Access C/C++ code from Python**
  - Convert function arguments from Python to C/C++
  - Make C/C++ function call using the converted values
  - Convert return values from C/C++ function to Python

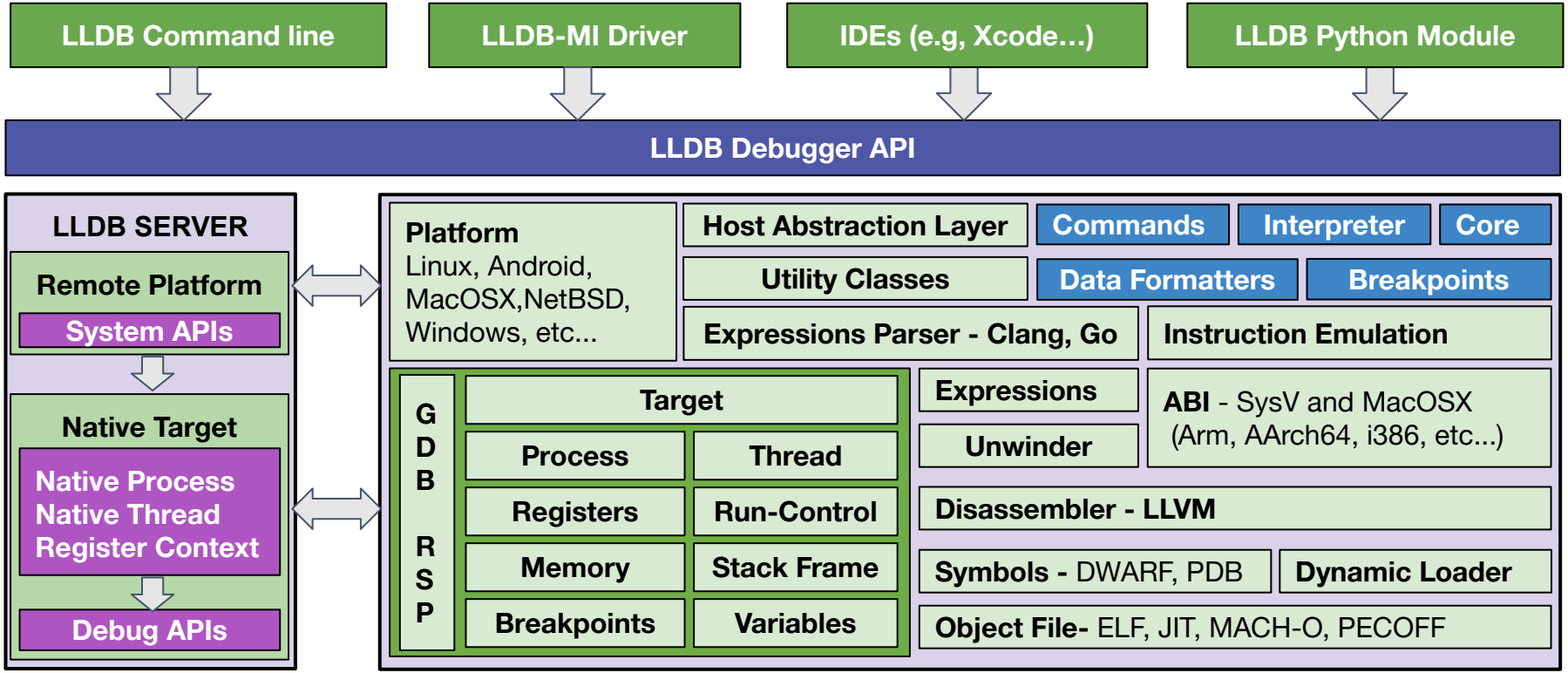


- **Embedding - Access Python functionality from C/C++ code**
  - **Type 1**: Execute a independent Python script from C/C++ code
  - **Type 2**: Execute Python script using data from C/C++ code
  - **Type 3**: An opposite of extending but similar too
    - Convert function arguments from C/C++ to Python
    - Make Python function call using the converted values
    - Convert return values from Python function to C/C++

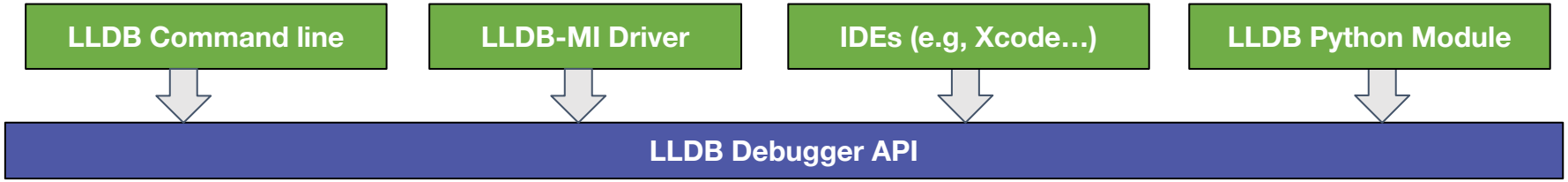
Extending or Embedding Python (Cont…)

- **Extending/Embedding - How to do it**
  - Write wrapper functions to access from both sides
  - Wrappers serve as a glue layer between languages
  - Need to convert function arguments from Python to C or vice versa
  - Need to return results from Python to C or vice versa
- **The problem**
  - Imagine doing this for a huge library containing hundreds of functions
  - Writing wrappers in C/C++ and Python for all functions to be exported
- **SWIG - Automatically generates Python interfaces**

LLDB Architecture



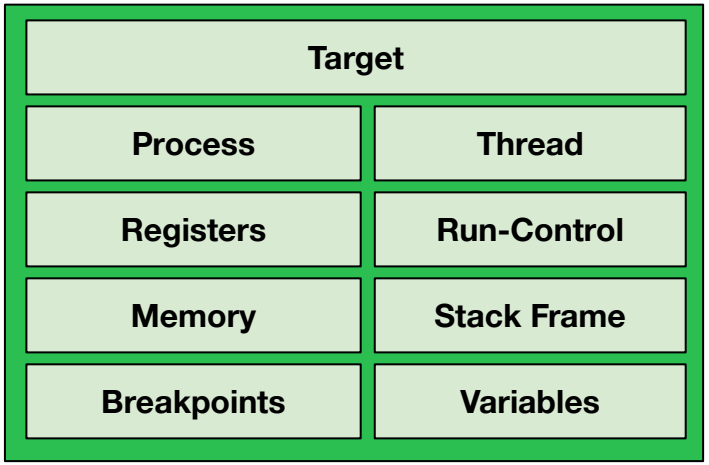
LLDB Debugger API



- A C++ shared library with object-oriented interface
- **LLDB.framework** on MacOS X and **lldb.so** on linux
- Used by:
  - **lldb -** The LLDB Debugger command line
  - **lldb-mi** and **lldbmi2 -** Machine Interface (MI) drivers
  - **XCode** and **Android Studio -** IDEs with graphical front-ends
  - **lldb Python module -** LLDB API exposed through python script bindings using SWIG
- **LLDB Python reference guide**
  - https://lldb.llvm.org/python_reference/index.html

lldb - Debuggee (or inferior) context on host system

- **Target - Inferior View**
  - Process Control
  - Thread Control
  - Registers and Memory
  - Stack Frames and Variables
  - Breakpoints and Watchpoints
  - Module loading



Python Representation - lldb module

- **SBDebugger**
  - main object that creates SBTargets and provides access to them
  - It also manages the overall debugging experiences
- **SBTarget:**
  - Represents the target program running under the debugger
- **SBProcess:**
  - Represents the process associated with the target program
- **SBThread**
  - Represents a thread of execution. SBProcess contains SBThread(s)
- **SBFrame**
  - Represents one of the stack frames associated with a thread
  - SBThread contains SBFrame(s).

Python Representation - lldb module (Cont...)

- **SBSymbolContext:**
  - A container that stores various debugger related info
- **SBValue:**
  - Represents the value of a variable, a register, or an expression
- **SBModule:**
  - Represents an executable image and its associated object and symbol files
  - SBTarget contains SBModule(s)
- **SBBreakpoint:**
  - Represents a logical breakpoint and its associated settings
  - SBTarget contains SBBreakpoint(s)

Setup LLDB on Ubuntu 16.04

- **Install LLDB + Clang on Ubuntu 16.04:**
  - sudo apt-get install clang-5.0 lldb-5.0
- **Create a command alias for clang & lldb:**
  - alias lldb=lldb-5.0
  - alias clang=clang-5.0
- **Checkout LLDB sources:**
  - **$** svn checkout https://llvm.org/svn/llvm-project/llvm/trunk llvm
  - **$** cd llvm/tools
  - **$** svn checkout https://llvm.org/svn/llvm-project/cfe/trunk clang
  - **$** svn checkout https://llvm.org/svn/llvm-project/lldb/trunk lldb
  - **$** cd ../..

Setup LLDB on Ubuntu 16.04

- **Building LLDB:**
  - **$** mkdir -p "build/host"
  - **$** cd build/host
  - **$** buildType=Release
  - **$** cmake -GNinja -DCMAKE_BUILD_TYPE="$buildType" ../../llvm -DCMAKE_ASM_COMPILER=clang-5.0 -DCMAKE_C_COMPILER=clang-5.0 -DCMAKE_CXX_COMPILER=clang++-5.0 "-DLLVM_TARGETS_TO_BUILD=X86"
  - **$** alias lldb=/home/omair/work/HKG18_LLDB_PYTHON/lldb-build/build/host/bin/lldb

Example: Python script as LLDB Command

Create C application using following code:

```
#include <stdio.h>
void func1() {
  printf("I am in function 1 !");}
void func2() {
  printf("I am in function 2 !");}

int main() {
  func1();
  func2();
  return 0;
}
```

Build app.c with debug symbols and without optimization:

clang -g -O0 -o app app.c

You now have an app.out which we can pass to lldb:

lldb -f app

Output:

(lldb) target create "app"

Current executable set to 'app' (x86_64).

Example: Python script as LLDB Command

Create a Python file test.py and paste the following script:

The script implements a lldb command test which sets a breakpoint at main, func1 and func2.

```
# This is test.py
import lldb
def btest(debugger, command, result, internal_dict):
  """ Just a test command to set a breakpoint """
  target = debugger.GetSelectedTarget()
  main_bp = target.BreakpointCreateByName("main")
  func1_bp = target.BreakpointCreateByName("func1")
  func2_bp = target.BreakpointCreateByName("func2")

def __lldb_init_module(debugger, internal_dict):
  debugger.HandleCommand('command script add -f btest.btest btest')
```

Thank You

#HKG18

HKG18 keynotes and videos on: connect.linaro.org

For further information: www.linaro.org