**GETTING STARTED**

Here's a short precis of how to run lldb if you are familiar with the gdb command set. We will start with some details on lldb command structure and syntax to help orient you.

**COMMAND STRUCTURE**

Unlike gdb's command set, which is rather free-form, we tried to make the lldb command syntax fairly structured. The commands are all of the form:

<noun> <verb> [-options [option-value]] [argument [argument...]]

The command line parsing is done before command execution, so it is uniform across all the commands. The command syntax for basic commands is very simple, arguments, options and option values are all white-space separated, and double-quotes are used to protect white-spaces in an argument. If you need to put a backslash or double-quote character in an argument you back-slash it in the argument. That makes the command syntax more regular, but it also means you may have to quote some arguments in lldb that you wouldn't in gdb.

Options can be placed anywhere on the command line, but if the arguments begin with a "-" then you have to tell lldb that you're done with options for the current command by adding an option termination: "--" So for instance if you want to launch a process and give the "process launch" command the "--stop-at-entry" option, yet you want the process you are about to launch to be launched with the arguments "-program\_arg value", you would type:

(lldb) process launch --stop-at-entry -- -program\_arg value

We also tried to reduce the number of special purpose argument parsers, which sometimes forces the user to be a little more explicit about stating their intentions. The first instance you'll note of this is the breakpoint command. In gdb, to set a breakpoint, you might enter

(gdb) break foo.c:12

to break at line 12 of foo.c, and:

(gdb) break foo

to break at the function foo. As time went on, the parser that tells foo.c:12 from foo fromfoo.c::foo (which means the function foo in the file foo.c) got more and more complex and bizarre, and especially in C++ there are times where there's really no way to specify the function you want to break on. The lldb commands are more verbose but also more precise and allow for intelligent auto completion.

To set the same file and line breakpoint in LLDB you can enter either of:

(lldb) breakpoint set --file foo.c --line 12   
(lldb) breakpoint set -f foo.c -l 12

To set a breakpoint on a function named foo in LLDB you can enter either of:

(lldb) breakpoint set --name foo   
(lldb) breakpoint set -n foo

You can use the --name option multiple times to make a breakpoint on a set of functions as well. This is convenient since it allows you to set commmon conditions or commands without having to specify them multiple times:

(lldb) breakpoint set --name foo --name bar

Setting breakpoints by name is even more specialized in LLDB as you can specify that you want to set a breakpoint at a function by method name. To set a breakpoint on all C++ methods namedfoo you can enter either of:

(lldb) breakpoint set --method foo   
(lldb) breakpoint set -M foo

To set a breakpoint Objective C selectors named alignLeftEdges: you can enter either of:

(lldb) breakpoint set --selector alignLeftEdges:   
(lldb) breakpoint set -S alignLeftEdges:

You can limit any breakpoints to a specific executable image by using the "--shlib <path>" ("-s <path>" for short):

(lldb) breakpoint set --shlib foo.dylib --name foo   
(lldb) breakpoint set -s foo.dylib -n foo

The --shlib option can also be repeated to specify several shared libraries.

Suggestions on more interesting primitives of this sort are also very welcome.

Just like gdb, the lldb command interpreter does a shortest unique string match on command names, so the following two commands will both execute the same command:

(lldb) breakpoint set -n "-[SKTGraphicView alignLeftEdges:]"   
(lldb) br s -n "-[SKTGraphicView alignLeftEdges:]"

lldb also supports command completion for source file names, symbol names, file names, etc. Completion is initiated by a hitting a **TAB**. Individual options in a command can have different completers, so for instance the "--file <path>" option in "breakpoint" completes to source files, the "--shlib <path>" option to currently loaded shared libraries, etc. We can even do things like if you specify "--shlib <path>", and are completing on "--file <path>", we will only list source files in the shared library specified by "--shlib <path>".

The individual commands are pretty extensively documented. You can use the help command to get an overview of which commands are available or to obtain details about specific commands. There is also an apropos command that will search the help text for all commands for a particular word and dump a summary help string for each matching command.

Finally, there is a mechanism to construct aliases for commonly used commands. So for instance if you get annoyed typing:

(lldb) breakpoint set --file foo.c --line 12

you can do:

(lldb) command alias bfl breakpoint set -f %1 -l %2   
(lldb) bfl foo.c 12

We have added a few aliases for commonly used commands (e.g. "step", "next" and "continue") but we haven't tried to be exhaustive because in our experience it is more convenient to make the basic commands unique down to a letter or two, and then learn these sequences than to fill the namespace with lots of aliases, and then have to type them all the way out.

However, users are free to customize lldb's command set however they like, and since lldb reads the file ~/.lldbinit at startup, you can store all your aliases there and they will be generally available to you. Your aliases are also documented in the help command so you can remind yourself of what you've set up.

One alias of note that we do include by popular demand is a weak emulator of gdb's "break" command. It doesn't try to do everything that gdb's break command does (for instance, it doesn't handle foo.c::bar. But it mostly works, and makes the transition easier. Also by popular demand, it is aliased to b. If you actually want to learn the lldb command set natively, that means it will get in the way of the rest of the breakpoint commands. Fortunately, if you don't like one of our aliases, you an easily get rid of it by running (for example):

(lldb) command unalias b

I actually also do:

(lldb) command alias b breakpoint

so I can run the native lldb breakpoint command with just b

The lldb command parser also supports "raw" commands, where, after command options are stripped off, the rest of the command string is passed uninterpreted to the command. This is convenient for commands whose arguments might be some complex expression that would be painful to backslash protect. For instance the "expression" command is a "raw" command for obvious reasons. The "help" output for a command will tell you if it is "raw" or not, so you know what to expect. The one thing you have to watch out for is that since raw commands still can have options, if your command string has dashes in it, you'll have to indicate these are not option markers by putting "--" after the command name, but before your command string.

lldb also has a built-in Python interpreter, which is accessible by the "script" command. All the functionality of the debugger is available as classes in the Python interpreter, so the more complex commands that in gdb you would introduce with the "define" command can be done by writing Python functions using the lldb-Python library, then loading the scripts into your running session and accessing them with the "script" command.

Having given an overview of lldb's command syntax, we proceed to lay out the stages of a standard debug session.

**LOADING A PROGRAM INTO LLDB**

First we need to set the program to debug. As with gdb, you can start lldb and specify the file you wish to debug on the command line:

$ lldb /Projects/Sketch/build/Debug/Sketch.app   
Current executable set to '/Projects/Sketch/build/Debug/Sketch.app' (x86\_64).

or you can specify it after the fact with the "file" command:

$ lldb   
(lldb) file /Projects/Sketch/build/Debug/Sketch.app   
Current executable set to '/Projects/Sketch/build/Debug/Sketch.app' (x86\_64).

**SETTING BREAKPOINTS**

We've discussed how to set breakpoints above. You can use help breakpoint set to see all the options for breakpoint setting. For instance, we might do:

(lldb) breakpoint set --selector alignLeftEdges:   
Breakpoint created: 1: name = 'alignLeftEdges:', locations = 1, resolved = 1

You can find out about the breakpoints you've set with:

(lldb) breakpoint list

Current breakpoints:

1: name = 'alignLeftEdges:', locations = 1, resolved = 1

1.1: where = Sketch`-[SKTGraphicView alignLeftEdges:] + 33 at /Projects/Sketch/SKTGraphicView.m:1405, address = 0x0000000100010d5b, resolved, hit count = 0

Note that setting a breakpoint creates a *logical* breakpoint, which could resolve to one or more*locations*. For instance, break by selector would set a breakpoint on all the methods that implement that selector in the classes in your program. Similarly, a file and line breakpoint might result in multiple locations if that file and line were inlined in different places in your code.

The logical breakpoint has an integer id, and it's locations have an id within their parent breakpoint (the two are joined by a ".", e.g. 1.1 in the example above.)

Also the logical breakpoints remain *live* so that if another shared library were to be loaded that had another implementation of the "alignLeftEdges:" selector, the new location would be added to breakpoint 1 (e.g. a "1.2" breakpoint would be set on the newly loaded selector).

The other piece of information in the breakpoint listing is whether the breakpoint location was*resolved* or not. A location gets resolved when the file address it corresponds to gets loaded into the program you are debugging. For instance if you set a breakpoint in a shared library that then gets unloaded, that breakpoint location will remain, but it will no longer be *resolved*.

One other thing to note for gdb users is that lldb acts like gdb with:

(gdb) set breakpoint pending on

That is, lldb will always make a breakpoint from your specification, even if it couldn't find any locations that match the specification. You can tell whether the expression was resolved or not by checking the locations field in "breakpoint list", and we report the breakpoint as "pending" when you set it so you can tell you've made a typo more easily, if that was indeed the reason no locations were found:

(lldb) breakpoint set --file foo.c --line 12   
Breakpoint created: 2: file ='foo.c', line = 12, locations = 0 (pending)   
WARNING: Unable to resolve breakpoint to any actual locations.

You can delete, disable, set conditions and ignore counts either on all the locations generated by your logical breakpoint, or on any one of the particular locations your specification resolved to. For instance if we wanted to add a command to print a backtrace when we hit this breakpoint we could do:

(lldb) breakpoint command add 1.1   
Enter your debugger command(s). Type 'DONE' to end.   
> bt   
> DONE

By default, the breakpoint command add command takes lldb command line commands. You can also specify this explicitly by passing the "--command" option. Use "--script" if you want to implement your breakpoint command using the Python script instead.

This is an convenient point to bring up another feature of the lldb command help. Do:

(lldb) help break command add   
Add a set of commands to a breakpoint, to be executed whenever the breakpoint is hit.   
  
Syntax: breakpoint command add <cmd-options> <breakpt-id>   
etc...

When you see arguments to commands specified in the Syntax in angle brackets like <breakpt-id>, that indicates that that is some common argument type that you can get further help on from the command system. So in this case you could do:

(lldb) help <breakpt-id>   
<breakpt-id> -- Breakpoint ID's consist major and minor numbers; the major   
etc...

**SETTING WATCHPOINTS**

In addition to breakpoints, you can use help watchpoint to see all the commands for watchpoint manipulations. For instance, we might do the following to watch a variable called 'global' for write operation, but only stop if the condition '(global==5)' is true:

(lldb) watch set var global

Watchpoint created: Watchpoint 1: addr = 0x100001018 size = 4 state = enabled type = w

declare @ '/Volumes/data/lldb/svn/ToT/test/functionalities/watchpoint/watchpoint\_commands/condition/main.cpp:12'

(lldb) watch modify -c '(global==5)'

(lldb) watch list

Current watchpoints:

Watchpoint 1: addr = 0x100001018 size = 4 state = enabled type = w

declare @ '/Volumes/data/lldb/svn/ToT/test/functionalities/watchpoint/watchpoint\_commands/condition/main.cpp:12'

condition = '(global==5)'

(lldb) c

Process 15562 resuming

(lldb) about to write to 'global'...

Process 15562 stopped and was programmatically restarted.

Process 15562 stopped and was programmatically restarted.

Process 15562 stopped and was programmatically restarted.

Process 15562 stopped and was programmatically restarted.

Process 15562 stopped

\* thread #1: tid = 0x1c03, 0x0000000100000ef5 a.out`modify + 21 at main.cpp:16, stop reason = watchpoint 1

frame #0: 0x0000000100000ef5 a.out`modify + 21 at main.cpp:16

13

14 static void modify(int32\_t &var) {

15 ++var;

-> 16 }

17

18 int main(int argc, char\*\* argv) {

19 int local = 0;

(lldb) bt

\* thread #1: tid = 0x1c03, 0x0000000100000ef5 a.out`modify + 21 at main.cpp:16, stop reason = watchpoint 1

frame #0: 0x0000000100000ef5 a.out`modify + 21 at main.cpp:16

frame #1: 0x0000000100000eac a.out`main + 108 at main.cpp:25

frame #2: 0x00007fff8ac9c7e1 libdyld.dylib`start + 1

(lldb) frame var global

(int32\_t) global = 5

(lldb) watch list -v

Current watchpoints:

Watchpoint 1: addr = 0x100001018 size = 4 state = enabled type = w

declare @ '/Volumes/data/lldb/svn/ToT/test/functionalities/watchpoint/watchpoint\_commands/condition/main.cpp:12'

condition = '(global==5)'

hw\_index = 0 hit\_count = 5 ignore\_count = 0

(lldb)

**STARTING OR ATTACHING TO YOUR PROGRAM**

To launch a program in lldb we use the "process launch" command or one of its built in aliases:

(lldb) process launch   
(lldb) run   
(lldb) r

You can also attach to a process by process ID or process name. When attaching to a process by name, lldb also supports the "--waitfor" option which waits for the next process that has that name to show up, and attaches to it

(lldb) process attach --pid 123   
(lldb) process attach --name Sketch   
(lldb) process attach --name Sketch --waitfor

After you launch or attach to a process, your process might stop somewhere:

(lldb) process attach -p 12345   
Process 46915 Attaching   
Process 46915 Stopped   
1 of 3 threads stopped with reasons:   
\* thread #1: tid = 0x2c03, 0x00007fff85cac76a, where = libSystem.B.dylib`\_\_getdirentries64 + 10, stop reason = signal = SIGSTOP, queue = com.apple.main-thread

Note the line that says "1 of 3 threads stopped with reasons:" and the lines that follow it. In a multi-threaded environment it is very common for more than one thread to hit your breakpoint(s) before the kernel actually returns control to the debugger. In that case, you will see all the threads that stopped for some interesting reason listed in the stop message.

**CONTROLLING YOUR PROGRAM**

After launching, we can continue until we hit our breakpoint. The primitive commands for process control all exist under the "thread" command:

(lldb) thread continue   
Resuming thread 0x2c03 in process 46915   
Resuming process 46915   
(lldb)

At present you can only operate on one thread at a time, but the design will ultimately support saying "step over the function in Thread 1, and step into the function in Thread 2, and continue Thread 3" etc. When we eventually support keeping some threads running while others are stopped this will be particularly important. For convenience, however, all the stepping commands have easy aliases. So "thread continue" is just "c", etc.

The other program stepping commands are pretty much the same as in gdb. You've got:

(lldb) thread step-in // The same as gdb's "step" or "s"

(lldb) thread step-over // The same as gdb's "next" or "n"

(lldb) thread step-out // The same as gdb's "finish" or "f"

By default, lldb does defined aliases to all common gdb process control commands ("s", "step", "n", "next", "finish"). If we have missed any, please add them to your ~/.lldbinit file using the "command alias" command.

lldb also supported the *step by instruction* versions:

(lldb) thread step-inst // The same as gdb's "stepi" / "si"

(lldb) thread step-over-inst // The same as gdb's "nexti" / "ni"

Finally, lldb has a *run until line or frame exit* stepping mode:

(lldb) thread until 100

This command will run the thread in the current frame till it reaches line 100 in this frame or stops if it leaves the current frame. This is a pretty close equivalent to gdb's "until" command.

A process, by default, will share the lldb terminal with the inferior process. When in this mode, much like when debugging with gdb, when the process is running anything you type will go to the STDIN of the inferior process. To interrupt your inferior program, type CTRL+C.

If you attach to a process, or launch a process with the "--no-stdin" option, the command interpreter is always available to enter commands. This might be a little disconcerting to gdb users when always have an (lldb) prompt. This allows you to set a breakpoint, etc without having to explicitly interrupt the program you are debugging:

(lldb) process continue   
(lldb) breakpoint set --name stop\_here

There are many commands that won't work while running, and the command interpreter should do a good job of letting you know when this is the case. If you find any instances where the command interpreter isn't doing its job, please file a bug. This way of operation will set us up for a future debugging mode called *thread centric debugging*. This mode will allow us to run all threads and only stop the threads that are at breakpoints or have exceptions or signals.

The commands that currently work while running include interrupting the process to halt execution ("process interrupt"), getting the process status ("process status"), breakpoint setting and clearing (" breakpoint [set|clear|enable|disable|list] ..."), and memory reading and writing (" memory [read|write] ...").

The question of disabling stdio when running brings up a good opportunity to show how to set debugger properties in general. If you always want to run in the --no-stdin mode, you can set this as a generic process property using the lldb "settings" command, which is equivalent to gdb's "set" command. For instance, in this case you would say:

(lldb) settings set target.process.disable-stdio true

Over time, gdb's "set command became a wilderness of disordered options, so that there were useful options that even experienced gdb users didn't know about because they were too hard to find. We tried to organize the settings hierarchically using the structure of the basic entities in the debugger. For the most part anywhere you can specify a setting on a generic entity (threads, for example) you can also apply the option to a particular instance, which can also be convenient at times. You can view the available settings with "settings list" and there is help on the settings command explaining how it works more generally.

**EXAMINING THREAD STATE**

Once you've stopped, lldb will choose a current thread, usually the one that stopped "for a reason", and a current frame in that thread (on stop this is always the bottom-most frame). Many the commands for inspecting state work on this current thread/frame.

To inspect the current state of your process, you can start with the threads:

(lldb) thread list

Process 46915 state is Stopped

\* thread #1: tid = 0x2c03, 0x00007fff85cac76a, where = libSystem.B.dylib`\_\_getdirentries64 + 10, stop reason = signal = SIGSTOP, queue = com.apple.main-thread

thread #2: tid = 0x2e03, 0x00007fff85cbb08a, where = libSystem.B.dylib`kevent + 10, queue = com.apple.libdispatch-manager

thread #3: tid = 0x2f03, 0x00007fff85cbbeaa, where = libSystem.B.dylib`\_\_workq\_kernreturn + 10

The \* indicates that Thread 1 is the current thread. To get a backtrace for that thread, do:

(lldb) thread backtrace

thread #1: tid = 0x2c03, stop reason = breakpoint 1.1, queue = com.apple.main-thread

frame #0: 0x0000000100010d5b, where = Sketch`-[SKTGraphicView alignLeftEdges:] + 33 at /Projects/Sketch/SKTGraphicView.m:1405

frame #1: 0x00007fff8602d152, where = AppKit`-[NSApplication sendAction:to:from:] + 95

frame #2: 0x00007fff860516be, where = AppKit`-[NSMenuItem \_corePerformAction] + 365

frame #3: 0x00007fff86051428, where = AppKit`-[NSCarbonMenuImpl performActionWithHighlightingForItemAtIndex:] + 121

frame #4: 0x00007fff860370c1, where = AppKit`-[NSMenu performKeyEquivalent:] + 272

frame #5: 0x00007fff86035e69, where = AppKit`-[NSApplication \_handleKeyEquivalent:] + 559

frame #6: 0x00007fff85f06aa1, where = AppKit`-[NSApplication sendEvent:] + 3630

frame #7: 0x00007fff85e9d922, where = AppKit`-[NSApplication run] + 474

frame #8: 0x00007fff85e965f8, where = AppKit`NSApplicationMain + 364

frame #9: 0x0000000100015ae3, where = Sketch`main + 33 at /Projects/Sketch/SKTMain.m:11

frame #10: 0x0000000100000f20, where = Sketch`start + 52

You can also provide a list of threads to backtrace, or the keyword "all" to see all threads:

(lldb) thread backtrace all

You can select the current thread, which will be used by default in all the commands in the next section, with the "thread select" command:

(lldb) thread select 2

where the thread index is just the one shown in the "thread list" listing.

**EXAMINING STACK FRAME STATE**

The most convenient way to inspect a frame's arguments and local variables is to use the "frame variable" command:

(lldb) frame variable   
self = (SKTGraphicView \*) 0x0000000100208b40   
\_cmd = (struct objc\_selector \*) 0x000000010001bae1   
sender = (id) 0x00000001001264e0   
selection = (NSArray \*) 0x00000001001264e0   
i = (NSUInteger) 0x00000001001264e0   
c = (NSUInteger) 0x00000001001253b0

As you see above, if you don't specify any variable names, all arguments and locals will be shown. If you call "frame variable" passing in the names of a particular local(s), only those variables will be printed. For instance:

(lldb) frame variable self   
(SKTGraphicView \*) self = 0x0000000100208b40

You can also pass in a path to some subelement of one of the available locals, and that sub-element will be printed. For instance:

(lldb) frame variable self.isa   
(struct objc\_class \*) self.isa = 0x0000000100023730

The "frame variable" command is not a full expression parser but it does support a few simple operations like &, \*, ->, [] (no overloaded operators). The array brackets can be used on pointers to treat pointers as arrays:

(lldb) frame variable \*self   
(SKTGraphicView \*) self = 0x0000000100208b40   
(NSView) NSView = {   
(NSResponder) NSResponder = {   
...   
  
(lldb) frame variable &self   
(SKTGraphicView \*\*) &self = 0x0000000100304ab   
  
(lldb) frame variable argv[0]   
(char const \*) argv[0] = 0x00007fff5fbffaf8 "/Projects/Sketch/build/Debug/Sketch.app/Contents/MacOS/Sketch"

The frame variable command will also perform "object printing" operations on variables (currently we only support ObjC printing, using the object's "description" method. Turn this on by passing the -o flag to frame variable:

(lldb) frame variable -o self (SKTGraphicView \*) self = 0x0000000100208b40 <SKTGraphicView: 0x100208b40>

You can select another frame to view with the "frame select" command

(lldb) frame select 9   
frame #9: 0x0000000100015ae3, where = Sketch`function1 + 33 at /Projects/Sketch/SKTFunctions.m:11

You can also move up and down the stack by passing the "--relative" ("-r") option. And we have built-in aliases "u" and "d" which behave like their gdb equivalents.

If you need to view more complex data or change program data, you can use the general "expression" command. It takes an expression and evaluates it in the scope of the currently selected frame. For instance:

(lldb) expr self   
$0 = (SKTGraphicView \*) 0x0000000100135430   
(lldb) expr self = 0x00   
$1 = (SKTGraphicView \*) 0x0000000000000000   
(lldb) frame var self   
(SKTGraphicView \*) self = 0x0000000000000000

You can also call functions:

(lldb) expr (int) printf ("I have a pointer 0x%llx.\n", self)   
$2 = (int) 22   
I have a pointer 0x0.

As I said above, "expression" is one of the "raw" commands. So you don't have to quote your whole expression, nor backslash protect quotes, etc...

Finally, the results of the expressions are stored in persistent variables (of the form $[0-9]+) that you can use in further expressions, like:

(lldb) expr self = $0   
$4 = (SKTGraphicView \*) 0x0000000100135430