Becoming a GDB Power User

Greg Law

Co-founder and CEO, Undo software

Recap of Lightning Talk

- TUI mode
- Python
- Reversible debugging

Python Pretty Printers

```
class MyPrinter(object):
    def __init__(self,val):
        self.val = val
    def to_string(self):
        return ( self.val['member'])

import gdb.printing
pp = gdb.printing.RegexpCollectionPrettyPrinter('mystruct')
pp.add_printer('mystruct', '^mystruct$', MyPrinter)
gdb.printing.register_pretty_printer( gdb.current_objfile(), pp)
```

.gdbinit

My ~/.gdbinit is nice and simple:

```
        set history save on
    set print pretty on
```

If you're funky, it's easy for weird stuff to happen.

Hint: have a project gdbinit with lots of stuff in it, and source that.

Multiprocess Debugging

Debug multiple ‘inferiors’ simultaneously

Add new inferiors

Follow fork/exec

Non-stop mode

Other threads continue while you're at the prompt

Breakpoints and watchpoints

`watch foo`	stop when foo is modified
`watch -l foo`	watch location
`rwatch foo`	stop when foo is read
`watch foo thread 3`	stop when thread 3 modifies foo
`watch foo if foo > 10`	stop when foo is > 10

thread apply

```
thread apply 1-4 print $sp
thread apply all backtrace
```

calling inferior functions

`call foo()` will call foo in your inferior

But beware, print may well do too, e.g.

```
print foo()
print foo+bar if C++
print errno
```

And beware, below will call **`strcpy()`** *and* **`malloc()`**!

```
call strcpy( buffer, “Hello, world!\n”)
```

Dynamic Printf

Use dprintf to put printf's in your code without recompiling, e.g.

```
dprintf mutex_lock,"m is %p m->magic is %u\n",m,m->magic
```

control how the printfs happen:

```
    set dprintf-style gdb|call|agent
    set dprintf-function fprintf
    set dprintf-channel mylog
```

Catchpoints

Catchpoints are like breakpoints but catch certain events, such as C++ exceptions

- e.g. **`catch catch`** to stop when C++ exceptions are caught
- e.g. **`catch syscall nanosleep`** to stop at nanosleep system call
- e.g. **`catch syscall 100`** to stop at system call number 100

More Python

Create your own commands

```
class my_command( gdb.Command):
    '''doc string'''
    def __init__( self):
        gdb.Command.__init__( self, 'my-command', gdb.COMMAND_NONE)
    def invoke( self, args, from_tty):
         do_bunch_of_python()
my_command()
```

Yet More Python

Hook certain kinds of events

```
def stop_handler( ev):
    print( 'stop event!')
    if isinstance( ev, gdb.SignalEvent):
        print( 'its a signal: ' + ev.stop_signal)

gdb.events.stop.connect( stop_handler)
```

Other cool things...

`tbreak`	temporary breakpoint
`rbreak`	reg-ex breakpoint
`command`	list of commands to be executed when breakpoint hit
`silent`	special command to suppress output on breakpoint hit
`save breakpoints`	save a list of breakpoints to a script
`save history`	save history of executed gdb commands
`info line foo.c:42`	show PC for line
`info line * $pc`	show line begin/end for current program counter

And finally...

- gcc's -g and -O are orthogonal; gcc -Og is optimised but doesn't mess up debug
- see also gdb dashboard on github