```
 1 #!/usr/bin/python
 2 # @lint-avoid-python-3-compatibility
   -imports
 3 #
 4 # uflow  Trace method execution flow
    in high-level languages.
 5 #        For Linux, uses BCC, eBPF.
 6 #
 7 # USAGE: uflow [-C CLASS] [-M METHOD
   ] [-v] {java,perl,php,python,ruby,
   tcl} pid
 8 #
 9 # Copyright 2016 Sasha Goldshtein
10 # Licensed under the Apache License
   , Version 2.0 (the "License")
11 #
12 # 27-Oct-2016   Sasha Goldshtein
   Created this.
13
14 from __future__ import
   print_function
15 import argparse
16 from bcc import BPF, USDT, utils
17 import ctypes as ct
18 import time
19 import os
20
21 languages = ["java", "perl", "php",
   "python", "ruby", "tcl"]
```

```
22
23 examples = """examples:
24     ./uflow -l java 185
                   # trace Java method
   calls in process 185
25     ./uflow -l ruby 134
                   # trace Ruby method
   calls in process 134
26     ./uflow -M indexOf -l java 185
        # trace only 'indexOf'-prefixed
    methods
27     ./uflow -C '<stdin>' -l python
   180 # trace only REPL-defined
   methods
28 """
29 parser = argparse.ArgumentParser(
30     description="Trace method
   execution flow in high-level
   languages.",
31     formatter_class=argparse.
   RawDescriptionHelpFormatter,
32     epilog=examples)
33 parser.add_argument("-l", "--
   language", choices=languages,
34     help="language to trace")
35 parser.add_argument("pid", type=int
   , help="process id to attach to")
36 parser.add_argument("-M", "--method"
   ,
```

```
37     help="trace only calls to
   methods starting with this prefix")
38 parser.add_argument("-C", "--class"
   , dest="clazz",
39     help="trace only calls to
   classes starting with this prefix")
40 parser.add_argument("-v", "--verbose
   ", action="store_true",
41     help="verbose mode: print the
   BPF program (for debugging purposes
   )")
42 parser.add_argument("--ebpf", action
   ="store_true",
43     help=argparse.SUPPRESS)
44 args = parser.parse_args()
45
46 usdt = USDT(pid=args.pid)
47
48 program = """
49 struct call_t {
50     u64 depth;                    //
   first bit is direction (0 entry, 1
   return)
51     u64 pid;                      // (
   tgid << 32) + pid from
   bpf_get_current...
52     char clazz[80];
53     char method[80];
54 };
```

```
55
56 BPF_PERF_OUTPUT(calls);
57 BPF_HASH(entry, u64, u64);
58 """
59
60 prefix_template = """
61 static inline bool prefix_%s(char *
   actual) {
62     char expected[] = "%s";
63     for (int i = 0; i < sizeof(
   expected) - 1; ++i) {
64         if (expected[i] != actual[i
   ]) {
65             return false;
66         }
67     }
68     return true;
69 }
70 """
71
72 if args.clazz:
73     program += prefix_template % ("
   class", args.clazz)
74 if args.method:
75     program += prefix_template % ("
   method", args.method)
76
77 trace_template = """
78 int NAME(struct pt_regs *ctx) {
```

```
79      u64 *depth, zero = 0, clazz = 0
    , method = 0 ;
80      struct call_t data = {};
81
82      READ_CLASS
83      READ_METHOD
84      bpf_probe_read_user(&data.clazz
    , sizeof(data.clazz), (void *)clazz
    );
85      bpf_probe_read_user(&data.
    method, sizeof(data.method), (void
     *)method);
86
87      FILTER_CLASS
88      FILTER_METHOD
89
90      data.pid =
    bpf_get_current_pid_tgid();
91      depth = entry.
    lookup_or_try_init(&data.pid, &zero
    );
92      if (!depth) {
93          depth = &zero;
94      }
95      data.depth = DEPTH;
96      UPDATE
97
98      calls.perf_submit(ctx, &data,
    sizeof(data));
```

```
 99     return 0;
100 }
101 """
102
103 def enable_probe(probe_name,
    func_name, read_class, read_method
    , is_return):
104     global program, trace_template
    , usdt
105     depth = "*depth + 1" if not
    is_return else "*depth | (1ULL <<
    63)"
106     update = "++(*depth);" if not
    is_return else "if (*depth) --(*
    depth);"
107     filter_class = "if (!
    prefix_class(data.clazz)) { return
    0; }" \
108                    if args.clazz
    else ""
109     filter_method = "if (!
    prefix_method(data.method)) {
    return 0; }" \
110                    if args.method
    else ""
111     program += trace_template.
    replace("NAME", func_name
    )                \
112                              .
```

```
112 replace("READ_CLASS", read_class
    )         \
113                                 .
    replace("READ_METHOD", read_method
    )       \
114                                 .
    replace("FILTER_CLASS",
    filter_class)     \
115                                 .
    replace("FILTER_METHOD",
    filter_method)   \
116                                 .
    replace("DEPTH", depth
    )                   \
117                                 .
    replace("UPDATE", update)
118     usdt.enable_probe_or_bail(
    probe_name, func_name)
119
120 usdt = USDT(pid=args.pid)
121
122 language = args.language
123 if not language:
124     language = utils.
    detect_language(languages, args.pid
    )
125
126 if language == "java":
127     enable_probe("method__entry", "
```

```
127 java_entry",
128                     "bpf_usdt_readarg(
    2, ctx, &clazz);",
129                     "bpf_usdt_readarg(
    4, ctx, &method);", is_return=False
    )
130     enable_probe("method__return",
    "java_return",
131                     "bpf_usdt_readarg(
    2, ctx, &clazz);",
132                     "bpf_usdt_readarg(
    4, ctx, &method);", is_return=True)
133 elif language == "perl":
134     enable_probe("sub__entry", "
    perl_entry",
135                     "bpf_usdt_readarg(
    2, ctx, &clazz);",
136                     "bpf_usdt_readarg(
    1, ctx, &method);", is_return=False
    )
137     enable_probe("sub__return", "
    perl_return",
138                     "bpf_usdt_readarg(
    2, ctx, &clazz);",
139                     "bpf_usdt_readarg(
    1, ctx, &method);", is_return=True)
140 elif language == "php":
141     enable_probe("function__entry"
    , "php_entry",
```

```
142                     "bpf_usdt_readarg(
    4, ctx, &clazz);",
143                     "bpf_usdt_readarg(
    1, ctx, &method);", is_return=False
    )
144     enable_probe("function__return"
    , "php_return",
145                     "bpf_usdt_readarg(
    4, ctx, &clazz);",
146                     "bpf_usdt_readarg(
    1, ctx, &method);", is_return=True)
147 elif language == "python":
148     enable_probe("function__entry"
    , "python_entry",
149                     "bpf_usdt_readarg(
    1, ctx, &clazz);",   # filename
    really
150                     "bpf_usdt_readarg(
    2, ctx, &method);", is_return=False
    )
151     enable_probe("function__return"
    , "python_return",
152                     "bpf_usdt_readarg(
    1, ctx, &clazz);",   # filename
    really
153                     "bpf_usdt_readarg(
    2, ctx, &method);", is_return=True)
154 elif language == "ruby":
155     enable_probe("method__entry", "
```

```
155 ruby_entry",
156                     "bpf_usdt_readarg(
    1, ctx, &clazz);",
157                     "bpf_usdt_readarg(
    2, ctx, &method);", is_return=False
    )
158     enable_probe("method__return",
    "ruby_return",
159                     "bpf_usdt_readarg(
    1, ctx, &clazz);",
160                     "bpf_usdt_readarg(
    2, ctx, &method);", is_return=True)
161     enable_probe("cmethod__entry",
    "ruby_centry",
162                     "bpf_usdt_readarg(
    1, ctx, &clazz);",
163                     "bpf_usdt_readarg(
    2, ctx, &method);", is_return=False
    )
164     enable_probe("cmethod__return"
    , "ruby_creturn",
165                     "bpf_usdt_readarg(
    1, ctx, &clazz);",
166                     "bpf_usdt_readarg(
    2, ctx, &method);", is_return=True)
167 elif language == "tcl":
168     enable_probe("proc__args", "
    tcl_entry",
169                     "",  # no class/
```

```
169 file info available
170                     "bpf_usdt_readarg(
    1, ctx, &method);", is_return=False
    )
171     enable_probe("proc__return", "
    tcl_return",
172                     "",  # no class/
    file info available
173                     "bpf_usdt_readarg(
    1, ctx, &method);", is_return=True)
174 else:
175     print("No language detected;
    use -l to trace a language.")
176     exit(1)
177
178 if args.ebpf or args.verbose:
179     if args.verbose:
180         print(usdt.get_text())
181     print(program)
182     if args.ebpf:
183         exit()
184
185 bpf = BPF(text=program,
    usdt_contexts=[usdt])
186 print("Tracing method calls in %s
    process %d... Ctrl-C to quit." %
187       (language, args.pid))
188 print("%-3s %-6s %-6s %-8s %s" % ("
    CPU", "PID", "TID", "TIME(us)", "
```

```
188 METHOD"))
189
190 class CallEvent(ct.Structure):
191     _fields_ = [
192         ("depth", ct.c_ulonglong),
193         ("pid", ct.c_ulonglong),
194         ("clazz", ct.c_char * 80),
195         ("method", ct.c_char * 80)
196         ]
197
198 start_ts = time.time()
199
200 def print_event(cpu, data, size):
201     event = ct.cast(data, ct.
    POINTER(CallEvent)).contents
202     depth = event.depth & (~(1 <<
    63))
203     direction = "<- " if event.
    depth & (1 << 63) else "-> "
204     print("%-3d %-6d %-6d %-8.3f %-
    40s" % (cpu, event.pid >> 32,
205         event.pid & 0xFFFFFFFF,
    time.time() - start_ts,
206         ("  " * (depth - 1)) +
    direction + \
207             event.clazz.decode('utf
    -8', 'replace') + "." + \
208             event.method.decode('
    utf-8', 'replace')))
```

```
209
210 bpf["calls"].open_perf_buffer(
    print_event)
211 while 1:
212     try:
213         bpf.perf_buffer_poll()
214     except KeyboardInterrupt:
215         exit()
216
```