WAD:

A Module for Converting Fatal Extension Errors into Python Exceptions

David M. Beazley
Department of Computer Science
University of Chicago
beazley@cs.uchicago.edu

March 6, 2001

Python Extension Building

A popular use of Python

- Hand-written extensions.
- FPIG
- pyfort
- SIP
- BPL
- CXX
- Extension Classes
- GRAD
- SWIG
- (Apologies to anyone I missed)

Extension building is fun

- Python as control language for C,C++, or Fortran.
- Rapid development and prototyping
- Nice user interfaces

But, debugging of extensions is problematic

- At the very least, it's annoying.

A Python Error

```
% python spam.py
Traceback (most recent call last):
  File "spam.py", line 15, in ?
    blah()
  File "spam.py", line 12, in blah
    bar()
  File "spam.py", line 9, in bar
    foo()
  File "spam.py", line 6, in foo
    spam(3)
  File "spam.py", line 3, in spam
    doh(n)
NameError: There is no variable named 'doh'
```

An Extension Error

```
% python spam.py
Segmentation Fault (core dumped)
%
```

or

```
% python spam.py
Bus Error (core dumped)
```

or

```
% python spam.py
Assertion failed: n > 0, file debug.c, line 54
Abort (core dumped)
%
```

Well, obviously something “bad” happened

Common Failure Modes

Uninitialized Data

- Improper initialization of libraries.
- Forgetting to call an initialization function?
- Calling functions in the wrong order?

Improper argument checking

- Passing of NULL pointers.
- Improper conversion of Python objects to C.

Failed assertions

- Library may make extensive use of assert().
- This is good, but it causes execution to abort.

Weird stuff

- Illegal instructions.
- Bus error. Memory alignment problems.

Math errors

- Floating point exception (SIGFPE).
- Of course, this only happens after 50 hours of computation.

GDB Traceback

```
(gdb) where
#0  0xff1d9bf0 in __sigprocmask () from /usr/lib/libthread.so.1
#1  0xff1ce628 in _resetsig () from /usr/lib/libthread.so.1
#2  0xff1cdd18 in _sigon () from /usr/lib/libthread.so.1
#3  0xff1d0e8c in _thrp_kill () from /usr/lib/libthread.so.1
#4  0xfee49b10 in raise () from /usr/lib/libc.so.1
#5  0xfee3512c in abort () from /usr/lib/libc.so.1
#6  0xfee353d0 in _assert () from /usr/lib/libc.so.1
#7  0xfeee13ec in abort_crash () from /u0/beazley/Projects/WAD/WAD/Test/./
debugmodule.so
#8  0xfeee28ec in _wrap_abort_crash ()
   from /u0/beazley/Projects/WAD/WAD/Test/./debugmodule.so
#9  0x281c8 in call_builtin (func=0x1cc4f0, arg=0x1f9424, kw=0x0) at ceval.c:2650
#10 0x28094 in PyEval_CallObjectWithKeywords (func=0x1cc4f0, arg=0x1f9424, kw=0x0)
    at ceval.c:2618
#11 0x26764 in eval_code2 (co=0x1d37e0, globals=0x0, locals=0x1d37cf, args=0x1cc4f0,
    argcount=1762552, kws=0x0, kwcount=0, defs=0x0, defcount=0, owner=0x0) at
ceval.c:1951
#12 0x263a0 in eval_code2 (co=0x1d3858, globals=0x0, locals=0x1cc4f0, args=0x19b1a4,
    argcount=1883008, kws=0x1d7318, kwcount=0, defs=0x0, defcount=0, owner=0x0)
    at ceval.c:1850
#13 0x263a0 in eval_code2 (co=0x1d3e50, globals=0x0, locals=0x19b1a4, args=0x1a7374,
    argcount=1883128, kws=0x0, kwcount=0, defs=0x0, defcount=0, owner=0x0) at
ceval.c:1850
#14 0x285e0 in call_function (func=0x1a73a4, arg=0x18f114, kw=0x0) at ceval.c:2772
#15 0x28080 in PyEval_CallObjectWithKeywords (func=0x1a73a4, arg=0x18f114, kw=0x0)
    at ceval.c:2616
#16 0x680b0 in builtin_apply (self=0x0, args=0x0) at bltinmodule.c:88
#17 0x281c8 in call_builtin (func=0x1910c8, arg=0x1f9b54, kw=0x0) at ceval.c:2650
#18 0x28094 in PyEval_CallObjectWithKeywords (func=0x1910c8, arg=0x1f9b54, kw=0x0)
    at ceval.c:2618
#19 0x26764 in eval_code2 (co=0x1f3948, globals=0x0, locals=0x1f38f0, args=0x1910c8,
    argcount=1733540, kws=0x0, kwcount=0, defs=0x0, defcount=0, owner=0x2436e4)
    at ceval.c:1951
#20 0x285e0 in call_function (func=0x24374c, arg=0x1a606c, kw=0x0) at ceval.c:2772
#21 0x28080 in PyEval_CallObjectWithKeywords (func=0x261414, arg=0x18f114, kw=0x0)
    at ceval.c:2616
#22 0x98064 in PythonCmd (clientData=0x1cc8e0, interp=0x20e658, argc=0,
argv=0xffbee060)
    at ./_tkinter.c:1274
#23 0xff122064 in TclInvokeStringCommand (clientData=0x278538, interp=0x20e658,
objc=1,
    objv=0x24ec84) at ./../generic/tclBasic.c:1752
#24 0xff13e98c in TclExecuteByteCode (interp=0x20e658, codePtr=0x2a0cd0)
    at ./../generic/tclExecute.c:845
#25 0xff122bf8 in Tcl_EvalObjEx (interp=0x20e658, objPtr=0x2370c8, flags=0)
    at ./../generic/tclBasic.c:2723
#26 0xff258220 in TkInvokeButton (butPtr=0x279188) at ./../generic/tkButton.c:1457
#27 0xff257698 in ButtonWidgetObjCmd (clientData=0x279188, interp=0x20e658, objc=2,
    objv=0x295e00) at ./../generic/tkButton.c:835
#28 0xff15e18c in EvalObjv (interp=0x20e658, objc=2, objv=0x295e00,
command=0xff182128 "",
    length=0, flags=262144) at ./../generic/tclParse.c:932
#29 0xff15e2b8 in Tcl_EvalObjv (interp=0x20e658, objc=2, objv=0x295e00,
flags=262144)
    at ./../generic/tclParse.c:1019
#30 0xff122928 in Tcl_EvalObjEx (interp=0x20e658, objPtr=0x2370e0, flags=262144)
    at ./../generic/tclBasic.c:2565
#31 0xff165544 in Tcl_UplevelObjCmd (dummy=0x1, interp=0x20e658, objc=1,
objv=0x24ec80)
    at ./../generic/tclProc.c:614
#32 0xff13e98c in TclExecuteByteCode (interp=0x20e658, codePtr=0x2a0b70)
    at ./../generic/tclExecute.c:845
#33 0xff122bf8 in Tcl_EvalObjEx (interp=0x20e658, objPtr=0x274d50, flags=0)
    at ./../generic/tclBasic.c:2723
#34 0xff165afc in TclObjInterpProc (clientData=0x1, interp=0x20e658, objc=0,
    objv=0xffbeebd8) at ./../generic/tclProc.c:1001
#35 0xff15e18c in EvalObjv (interp=0x20e658, objc=2, objv=0xffbeebd8,
    command=0xffbef024 "\n    tkButtonUp .1907556\n", length=25, flags=0)
    at ./../generic/tclParse.c:932
#36 0xff15e7d0 in Tcl_EvalEx (interp=0x20e658,
    script=0xffbef024 "\n    tkButtonUp .1907556\n", numBytes=25, flags=-4264800)
    at ./../generic/tclParse.c:1393
#37 0xff15e9c0 in Tcl_Eval (interp=0x20e658,
    string=0xffbef024 "\n    tkButtonUp .1907556\n") at ./../generic/tclParse.c:1512
#38 0xff1243d0 in Tcl_GlobalEval (interp=0x20e658,
    command=0xffbef024 "\n    tkButtonUp .1907556\n") at ./../generic/tclBasic.c:4139
#39 0xff221a40 in Tk_BindEvent (bindingTable=0xffbef024, eventPtr=0x29ffa0,
    tkwin=0x2790a8, numObjects=2045728, objectPtr=0xffbef170) at ./../generic/
tkBind.c:1784
#40 0xff226450 in TkBindEventProc (winPtr=0x2790a8, eventPtr=0x29ffa0)
    at ./../generic/tkCmds.c:244
#41 0xff22c218 in Tk_HandleEvent (eventPtr=0x29ffa0) at ./../generic/tkEvent.c:737
#42 0xff22c61c in WindowEventProc (evPtr=0x29ff98, flags=-1) at ./../generic/
tkEvent.c:1072
#43 0xff15bb54 in Tcl_ServiceEvent (flags=-1) at ./../generic/tclNotify.c:607
#44 0xff15beec in Tcl_DoOneEvent (flags=-1) at ./../generic/tclNotify.c:846
#45 0x99314 in EventHook () at ./_tkinter.c:2020
#46 0xbaf30 in rl_read_key () at input.c:374
#47 0xac920 in readline_internal_char () at readline.c:454
#48 0xaca64 in readline_internal_charloop () at readline.c:507
#49 0xaca94 in readline_internal () at readline.c:521
#50 0xac704 in readline (prompt=0x1cbd9c ">>> ") at readline.c:349
#51 0x8249c in call_readline (prompt=0x1cbd9c ">>> ") at ./readline.c:462
#52 0x21ae0 in PyOS_Readline (prompt=0x1cbd9c ">>> ") at myreadline.c:118
#53 0x205a0 in tok_nextc (tok=0x27abd0) at tokenizer.c:192
#54 0x20fb4 in PyTokenizer_Get (tok=0x27abd0, p_start=0xffbef8c4, p_end=0xffbef8c0)
    at tokenizer.c:516
#55 0x20274 in parsetok (tok=0x27abd0, g=0x17026c, start=256, err_ret=0xffbef9b0)
    at parsetok.c:128
#56 0x20158 in PyParser_ParseFile (fp=0x18ebe8, filename=0xbf628 "<stdin>",
g=0x17026c,
    start=256, ps1=0x1cbd9c ">>> ", ps2=0x25a7e4 "... ", err_ret=0xffbef9b0)
    at parsetok.c:75
#57 0x3a9c0 in PyRun_InteractiveOne (fp=0x18ebe8, filename=0xbf628 "<stdin>")
    at pythonrun.c:514
#58 0x3a8bc in PyRun_InteractiveLoop (fp=0x18ebe8, filename=0xbf628 "<stdin>")
    at pythonrun.c:478
#59 0x3a7ac in PyRun_AnyFileEx (fp=0x18ebe8, filename=0xbf628 "<stdin>", closeit=0)
    at pythonrun.c:453
#60 0x3a76c in PyRun_AnyFile (fp=0x18ebe8, filename=0xbf628 "<stdin>") at
pythonrun.c:444
#61 0x1ff20 in Py_Main (argc=3, argv=0xffbefc74) at main.c:297
#62 0x1f90c in main (argc=3, argv=0xffbefc74) at python.c:10
(gdb)
```

GDB Traceback

```
(gdb) where
#0  0xff1d9bf0 in __sigprocmask () from /usr/lib/libthread.so.1
#1  0xff1ce628 in _resetsig () from /usr/lib/libthread.so.1
#2  0xff1cdd18 in _sigon () from /usr/lib/libthread.so.1
#3  0xff1d0e8c in _thrp_kill () from /usr/lib/libthread.so.1
#4  0xfee49b10 in raise () from /usr/lib/libc.so.1
#5  0xfee3512c in abort () from /usr/lib/libc.so.1
#6  0xfee353d0 in _assert () from /usr/lib/libc.so.1
#7  0xfeee13ec in abort_crash () from /u0/beazley/Projects/WAD/WAD/Test/./
debugmodule.so
#8  0xfeee28ec in _wrap_abort_crash ()
   from /u0/beazley/Projects/WAD/WAD/Test/./debugmodule.so
#9  0x281c8 in call_builtin (func=0x1cc4f0, arg=0x1f9424, kw=0x0) at ceval.c:2650
#10 0x28094 in PyEval_CallObjectWithKeywords (func=0x1cc4f0, arg=0x1f9424, kw=0x0)
    at ceval.c:2618
#11 0x26764 in eval_code2 (co=0x1d37e0, globals=0x0, locals=0x1d37cf, args=0x1cc4f0,
    argcount=1762552, kws=0x0, kwcount=0, defs=0x0, defcount=0, owner=0x0) at
ceval.c:1951
#12 0x263a0 in eval
    argcount=188300
    at ceval.c:185
#13 0x263a0 in eval
    argcount=188312
ceval.c:1850
#14 0x285e0 in cal
#15 0x28080 in PyEv
    at ceval.c:2616
#16 0x680b0 in bui
#17 0x281c8 in cal
#18 0x28094 in PyEv
    at ceval.c:2618
#19 0x26764 in eval
    argcount=173354
    at ceval.c:195
#20 0x285e0 in cal
#21 0x28080 in PyEv
    at ceval.c:2616
#22 0x98064 in Pyth
argv=0xffbee060)
    at ./_tkinter.c
#23 0xff122064 in T
objc=1,
    objv=0x24ec84)
#24 0xff13e98c in T
    at ./../generic
#25 0xff122bf8 in T
    at ./../generic
#26 0xff258220 in T
#27 0xff257698 in E
    objv=0x295e00) at ./../generic/tkButton.c:835
#28 0xff15e18c in EvalObjv (interp=0x20e658, objc=2, objv=0x295e00,
command=0xff182128 "",
    length=0, flags=262144) at ./../generic/tclParse.c:932
#29 0xff15e2b8 in Tcl_EvalObjv (interp=0x20e658, objc=2, objv=0x295e00,
flags=262144)
    at ./../generic/tclParse.c:1019
#30 0xff122928 in Tcl_EvalObjEx (interp=0x20e658, objPtr=0x2370e0, flags=262144)
    at ./../generic/tclBasic.c:2565
#31 0xff165544 in Tcl_UplevelObjCmd (dummy=0x1, interp=0x20e658, objc=1,
objv=0x24ec80)
    at ./../generic/tclProc.c:614
#32 0xff13e98c in TclExecuteByteCode (interp=0x20e658, codePtr=0x2a0b70)
    at ./../generic/tclExecute.c:845
#33 0xff122bf8 in Tcl_EvalObjEx (interp=0x20e658, objPtr=0x274d50, flags=0)
    at ./../generic/tclBasic.c:2723
#34 0xff165afc in TclObjInterpProc (clientData=0x1, interp=0x20e658, objc=0,
    objv=0xffbeebd8) at ./../generic/tclProc.c:1001
#35 0xff15e18c in EvalObjv (interp=0x20e658, objc=2, objv=0xffbeebd8,
    command=0xffbef024 "\n    tkButtonUp .1907556\n", length=25, flags=0)
    at ./../generic/tclParse.c:932
#36 0xff15e7d0 in Tcl_EvalEx (interp=0x20e658,
    script=0xffbef024 "\n    tkButtonUp .1907556\n", numBytes=25, flags=-4264800)
    at ./../generic/tclParse.c:1393
#37 0xff15e9c0 in Tcl_Eval (interp=0x20e658,
    string=0xffbef024 "\n    tkButtonUp .1907556\n") at ./../generic/tclParse.c:1512
                                                                                39
                                                                                )
    at pythonrun.c:478
#59 0x3a7ac in PyRun_AnyFileEx (fp=0x18ebe8, filename=0xbf628 "<stdin>", closeit=0)
    at pythonrun.c:453
#60 0x3a76c in PyRun_AnyFile (fp=0x18ebe8, filename=0xbf628 "<stdin>") at
pythonrun.c:444
#61 0x1ff20 in Py_Main (argc=3, argv=0xffbefc74) at main.c:297
#62 0x1f90c in main (argc=3, argv=0xffbefc74) at python.c:10
(gdb)
```


```
(gdb) where
#0  0xff1d9bf0 in __sigprocmask () from /usr/lib/libthread.so.1
#1  0xff1ce628 in _resetsig () from /usr/lib/libthread.so.1
#2  0xff1cdd18 in _sigon () from /usr/lib/libthread.so.1
#3  0xff1d0e8c in _thrp_kill () from /usr/lib/libthread.so.1
#4  0xfee49b10 in raise () from /usr/lib/libc.so.1
#5  0xfee3512c in abort () from /usr/lib/libc.so.1
#6  0xfee353d0 in _assert () from /usr/lib/libc.so.1
#7  0xfeee13ec in abort_crash () from /u0/beazley/Projects/WAD/WAD/Test/
./debugmodule.so
#8  0xfeee28ec in _wrap_abort_crash ()
   from /u0/beazley/Projects/WAD/WAD/Test/./debugmodule.so
#9  0x281c8 in call_builtin (func=0x1cc4f0, arg=0x1f9424, kw=0x0) at
ceval.c:2650
#10 0x28094 in PyEval_CallObjectWithKeywords (func=0x1cc4f0,
arg=0x1f9424, kw=0x0)
    at ceval.c:2618
```

Debugging Problems

General problem

- Traditional debugger doesn't know anything about Python scripts.
- Mostly provides information about the implementation of Python.
- Can't fully answer question of "how did I get here?"
- A problem if you have a lot of Python code.

Sometimes it is hard to reproduce a problem

- Run-time environment may be complex.
- Problems may be due to timing or precise event sequences.
- Problem may only occur after a long period of time.

Other issues

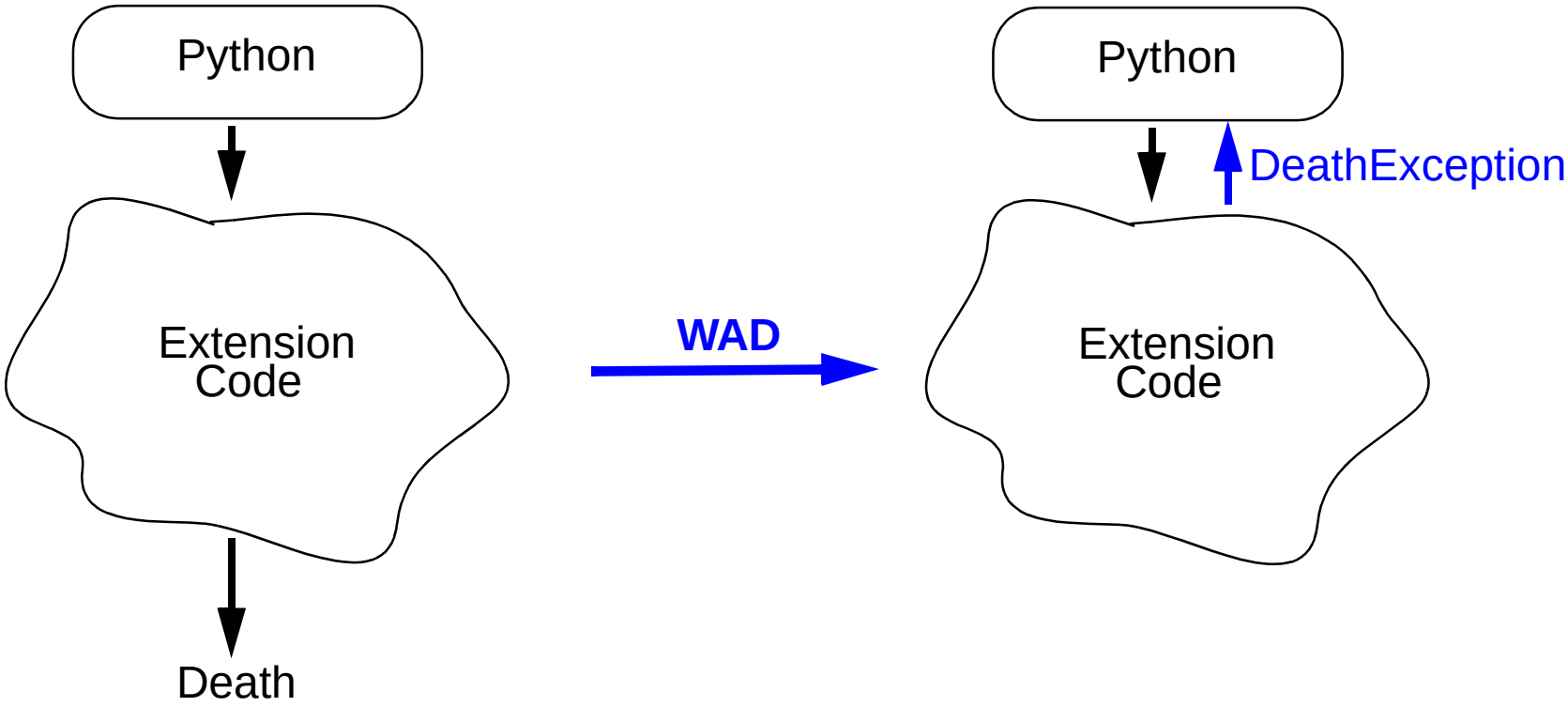
- Requires users to run a separate application (very unpython).
- Requires users to have a C development environment installed.
- Assumes users know how to use the C debugger.

Claim: I think you can do better

WAD

Wrapped Application Debugger

- Idea: Maybe you could turn fatal extension errors into Python Exceptions
- Seg faults, bus errors, illegal instructions, failed assertions, and math errors.



Demo

WAD Demo

```
% python
>>> import debug
>>> debug.seg_crash()
Segmentation fault (core dumped)
%
% python
>>> import debug
>>> import libwadpy
WAD Enabled
>>> debug.seg_crash()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
SegFault: [ C stack trace ]

#2   0x000281c0 in call_builtin(func=0x1cbaf0,arg=0x18f114,kw=0x0) in 'ceval.c', line
2650
#1   0xfeee26b8 in _wrap_seg_crash(self=0x0,args=0x18f114) in 'pydebug.c', line 510
#0   0xfeee1258 in seg_crash(0x1,0xfeef2d48,0x19a9f8,0x0,0x7365675f,0x5f5f6469) in
'debug.c', line 18

/u0/beazley/Projects/WAD/WAD/Test/debug.c, line 18

    int seg_crash() {
      int *a = 0;
 =>   *a = 3;
      return 1;
    }
```

Big Picture

WAD

- WAD is a dynamically loadable Python extension module.
- Converts catastrophic errors to Python exceptions.

Key features

- No modifications to Python
- No modifications to extensions.
- No recompilation.
- No relinking.
- No separate debugger required (gdb, dbx, etc.)
- No C, C++, Fortran development environment needed.
- No added performance penalty.

The rest of this talk

- Using WAD
- Gory implementation details
- Limitations
- Future directions.

Using WAD

1. Explicit import

```
import libwadpy
```

2. Implicit linking

```
ld -shared $(OBJS) -o foomodule.so -lwadpy
```

- Automatically loads WAD when the extension is loaded.

What WAD provides

- 4 new Python exceptions (SegFault, BusError, AbortError, IllegalInstruction)
- Exceptions are added to __builtin__ module.
- A new Python type (WadObject). Returned as an exception value.
- Otherwise, no public functions, constants, or variables (libwadpy is empty).
- Also: WAD is completely self contained

Exception Handling with WAD

Just like ordinary Python exception handling

- Except that you get a much more interesting exception value

```
try:
    naughty bits
except SegFault, s:
    t = s.args[0]       # Get trace object
    print t             # Prints stack trace
    len(t)              # Number of stack frames
    f = s[3]            # Returns a stack frame
    f.__FILE__          # Source file
    f.__LINE__          # Source line
    f.__EXE__           # Object file
    f.__PC__            # Program counter
    f.__STACK__         # Raw stack frame
    ...
    f.name              # Value of parameter or local name
```

Implementation Overview

Unix signal handling

- SIGSEGV
- SIGBUS
- SIGABRT
- SIGFPE
- SIGILL

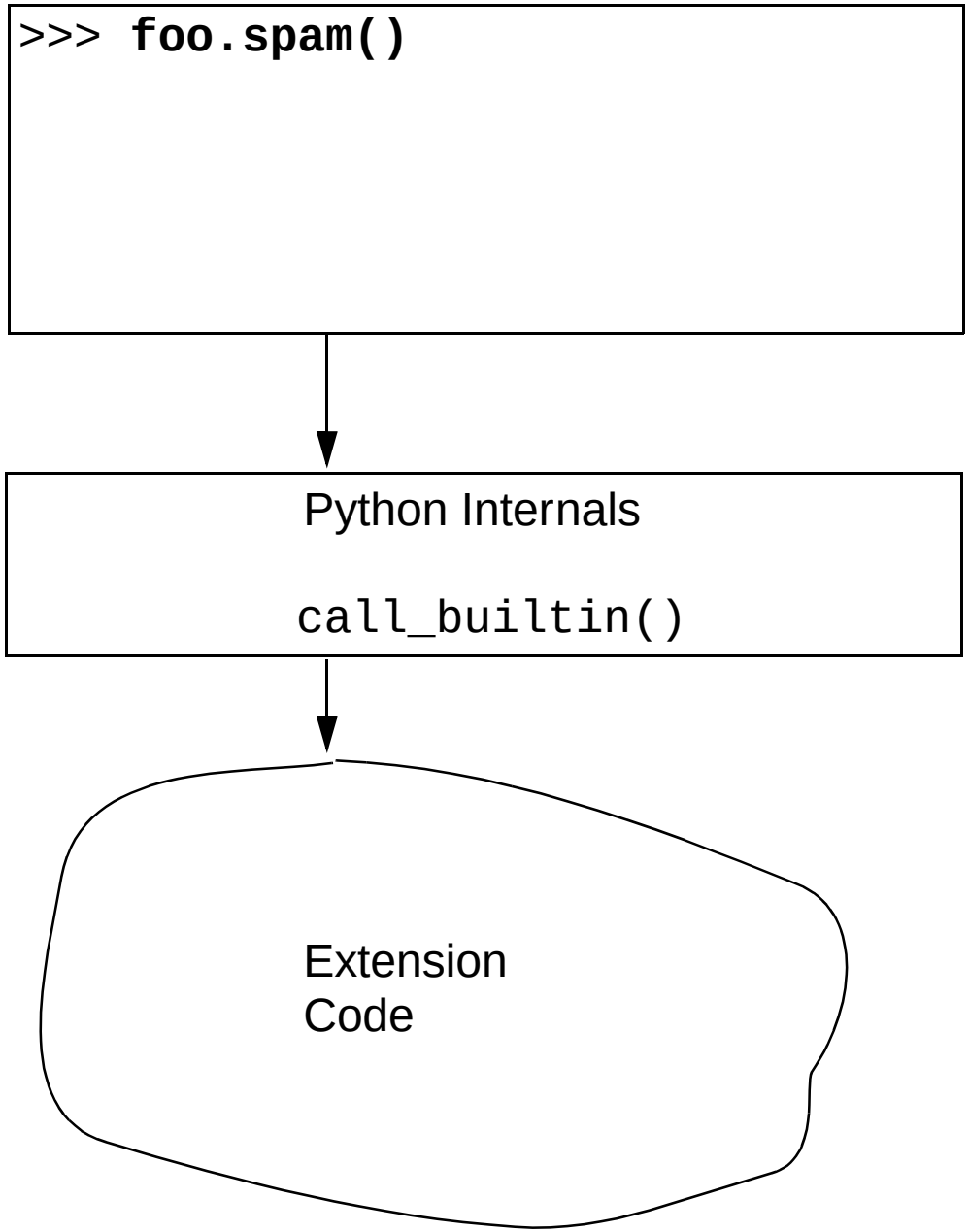
Process introspection

- Discovering program context.
- Reading of object files
- Collection of debugging data

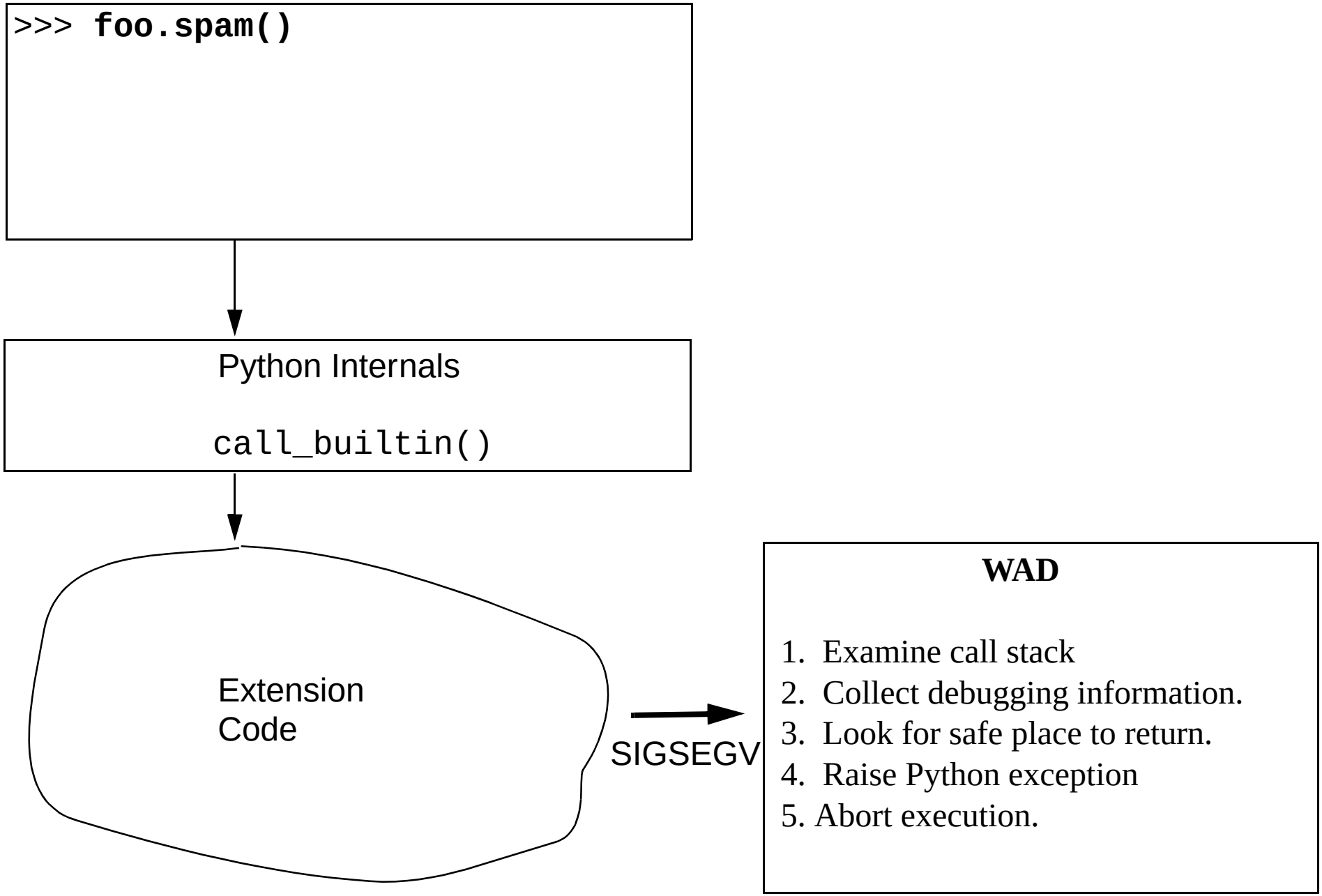
Abort and return to Python

- How do you actually get back to the interpreter?

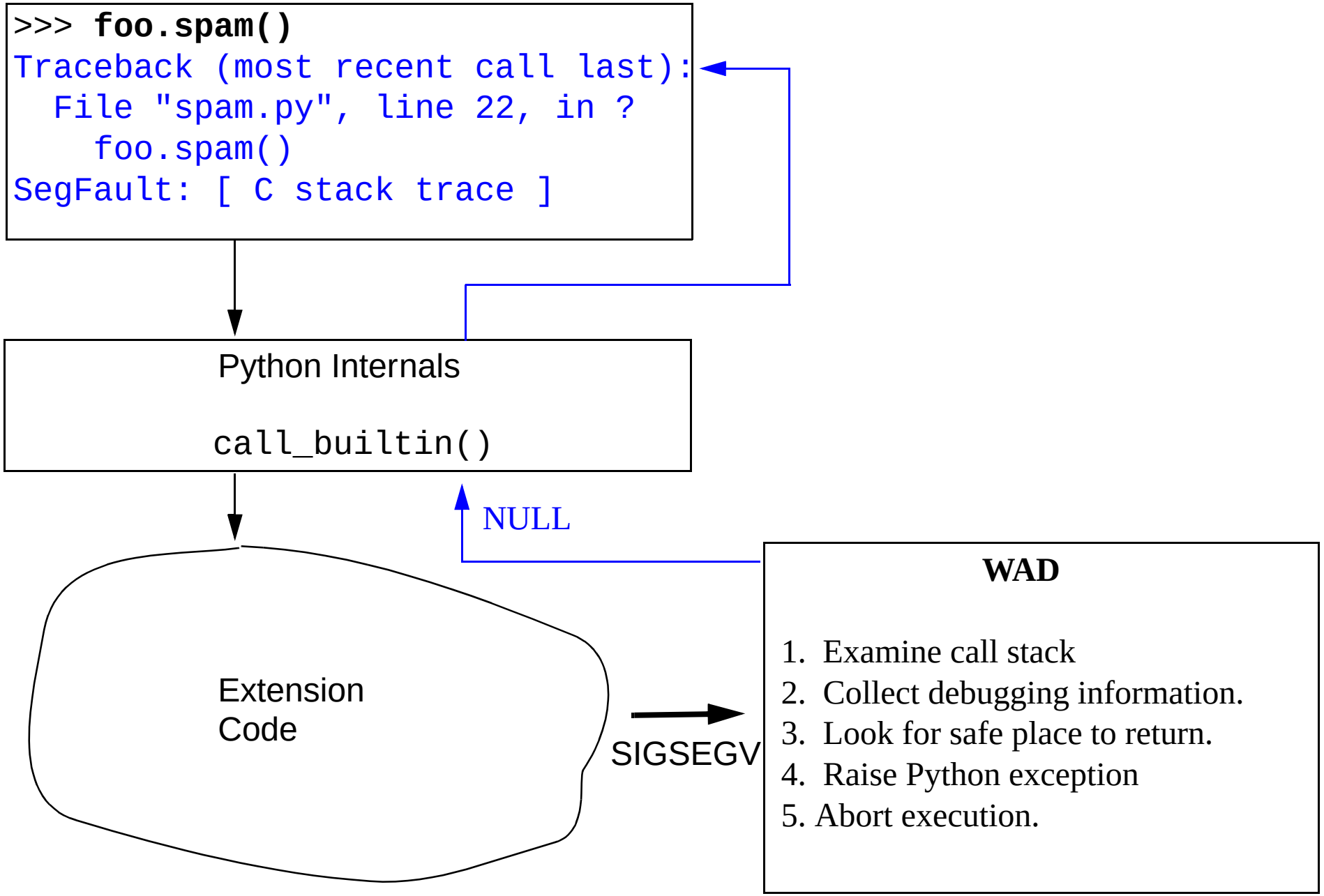
Control Flow



Control Flow (cont)

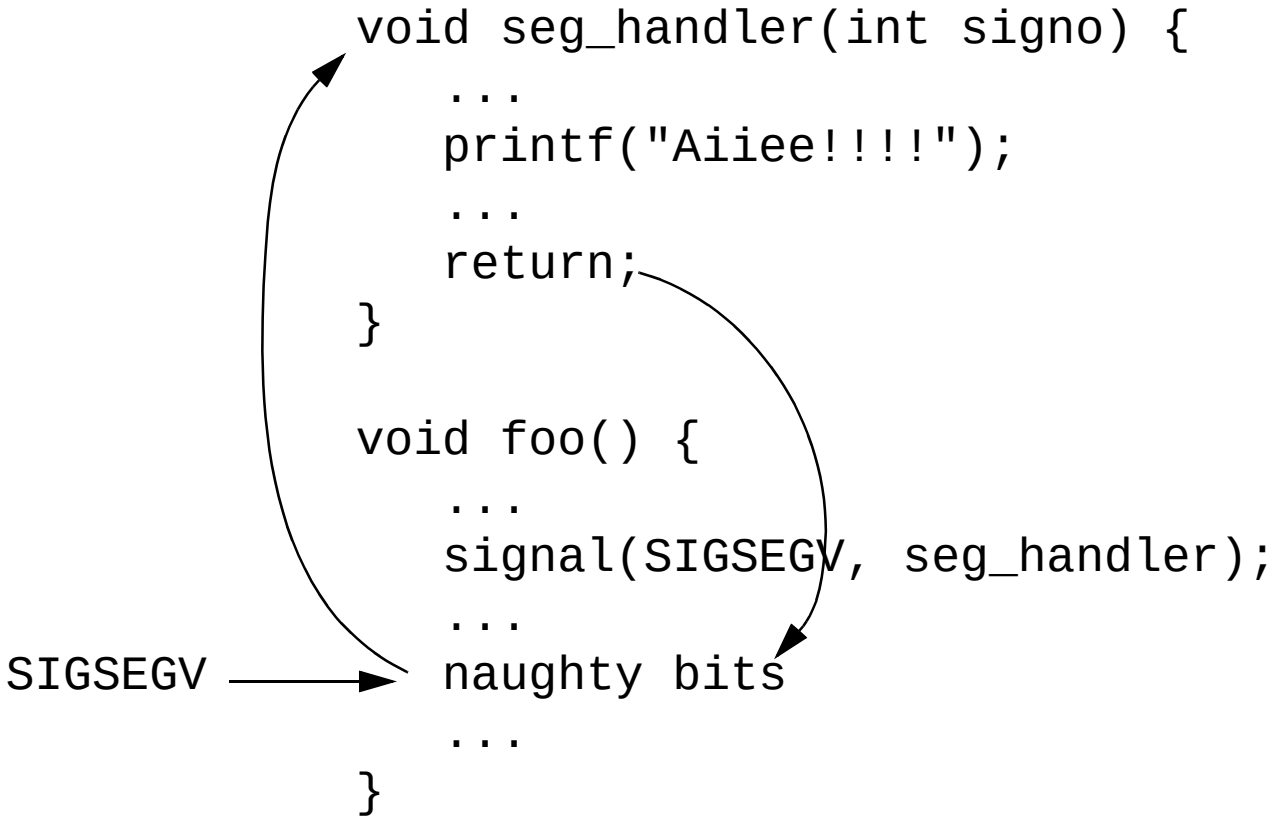


Control Flow (cont)



Signal Handling

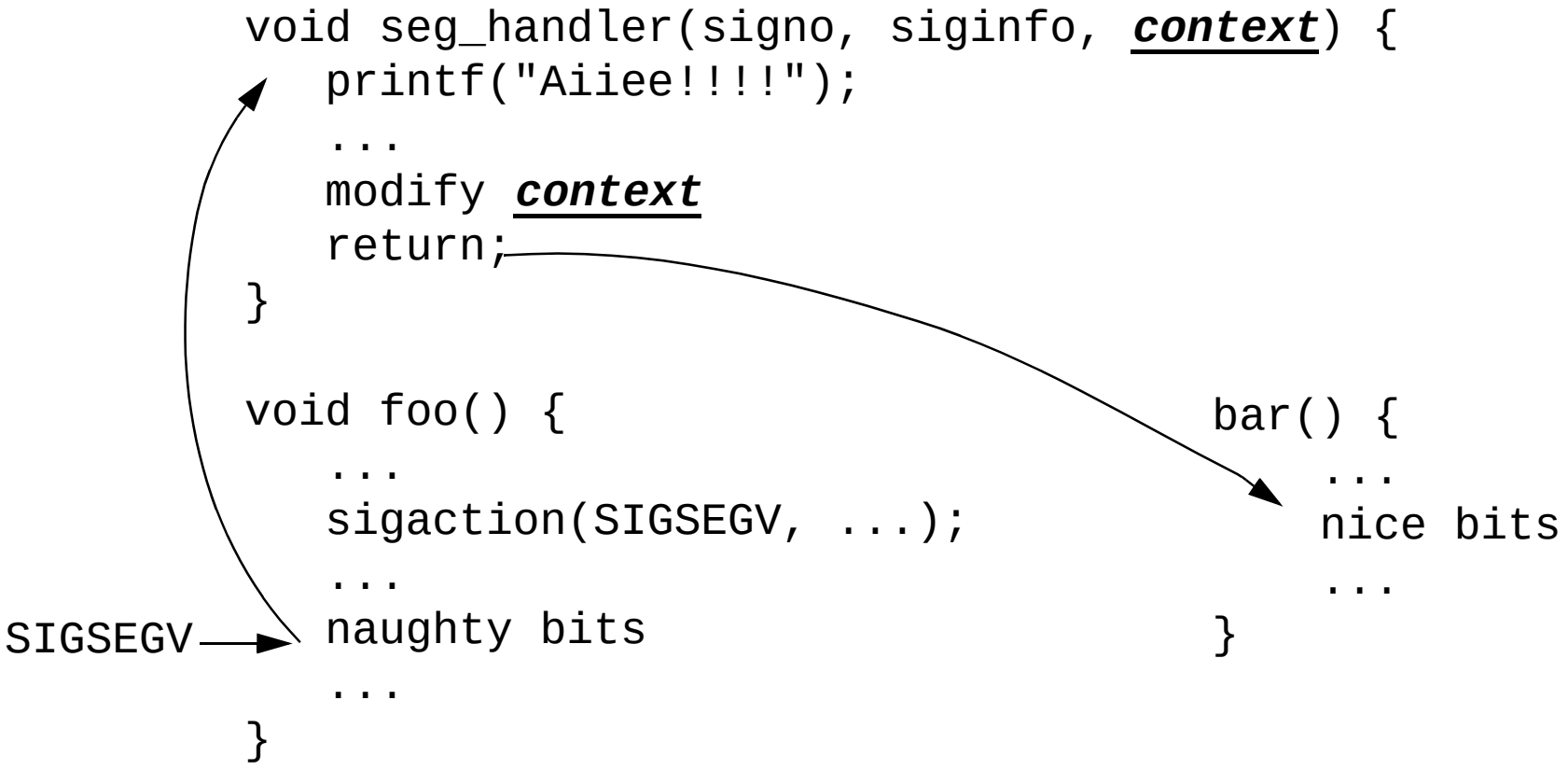
Traditional Signal Handling



- Signal handler executes on error.
- Unfortunately, execution resumes at point of error (and repeats).
- Note: Python signal module can’t handle SEGSEGV and related signals.

Signal Handling

Advanced Signal Handling



- Rarely used form of sigaction() allows signal handler to modify context
- Includes all CPU registers, program counter (PC), stack pointer (SP)
- Changes take effect on return from signal handler.
- Normally used to implement user-level thread libraries.

WAD: In a Nutshell

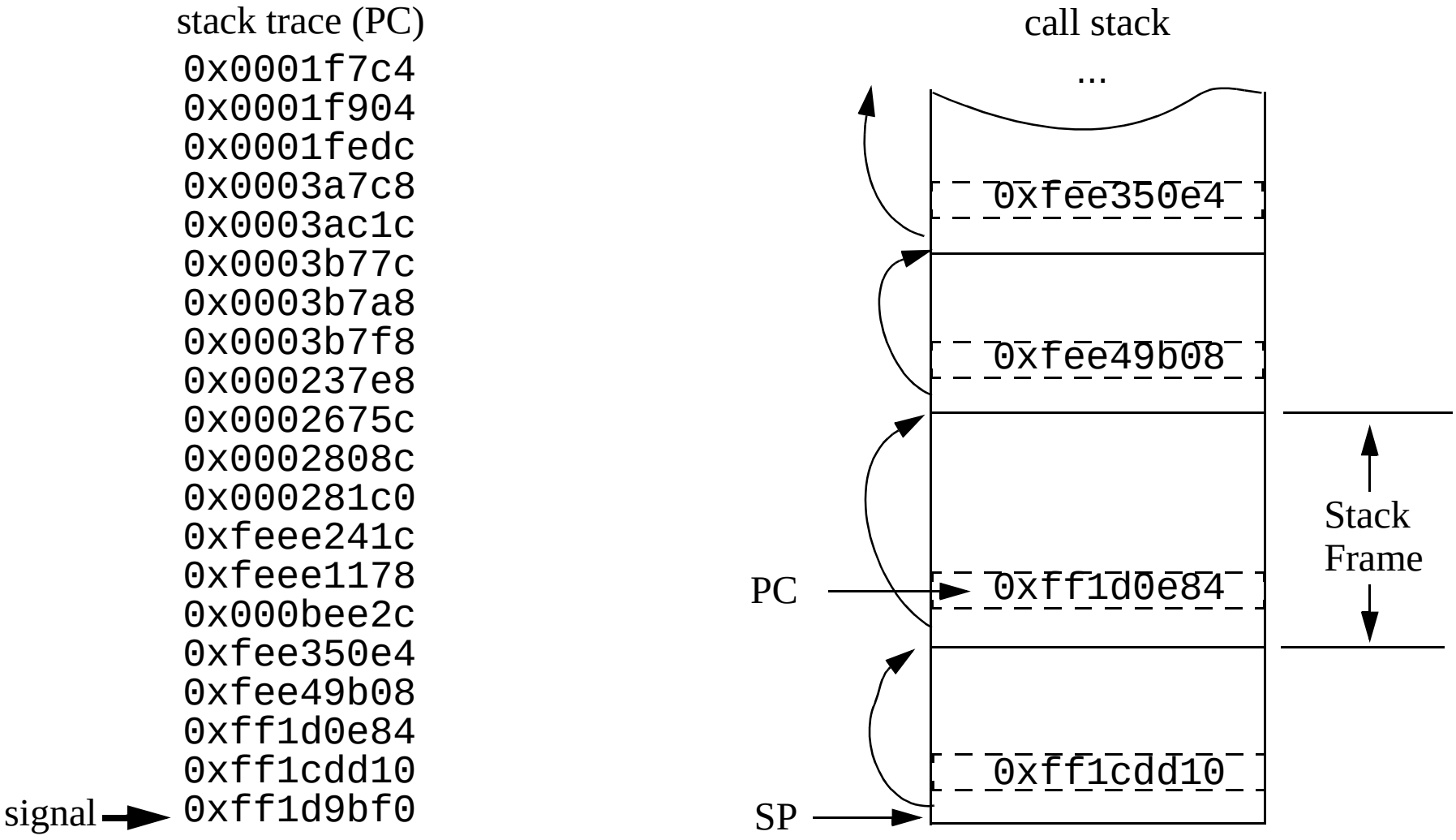
Signal handling + context rewriting

- Signal handler collects process information.
- Raise Python exception.
- Rewrite process context so that Python interpreter regains control.
- Return from signal handler.

Issues

- How do you perform process introspection?
- How do you figure out where to return in Python?
- How do you abort execution without breaking the universe?

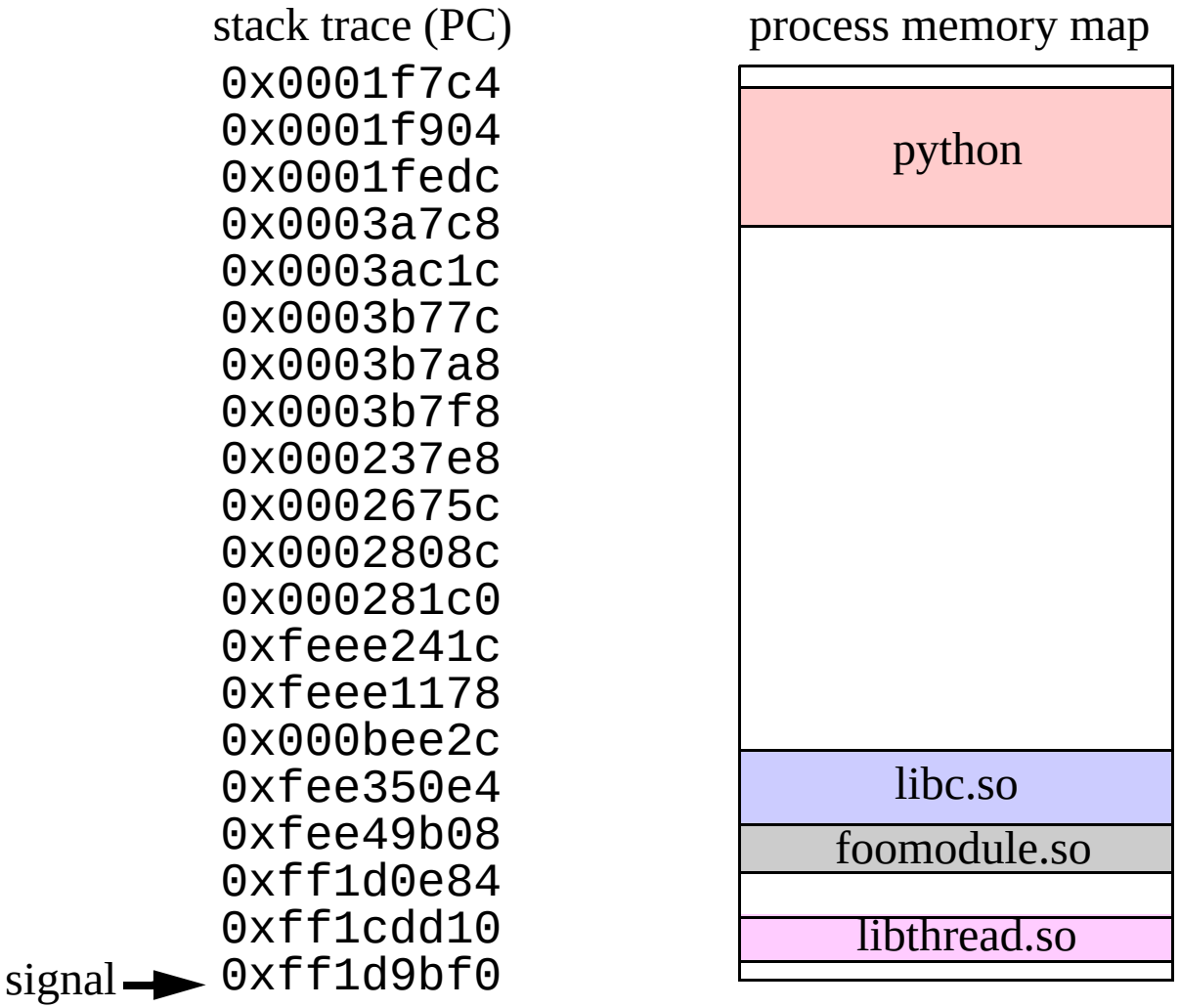
Finding Program Context



1. Generate raw stack trace

- A very simple while loop.
- Get sequence of PC, SP values and stack frames.

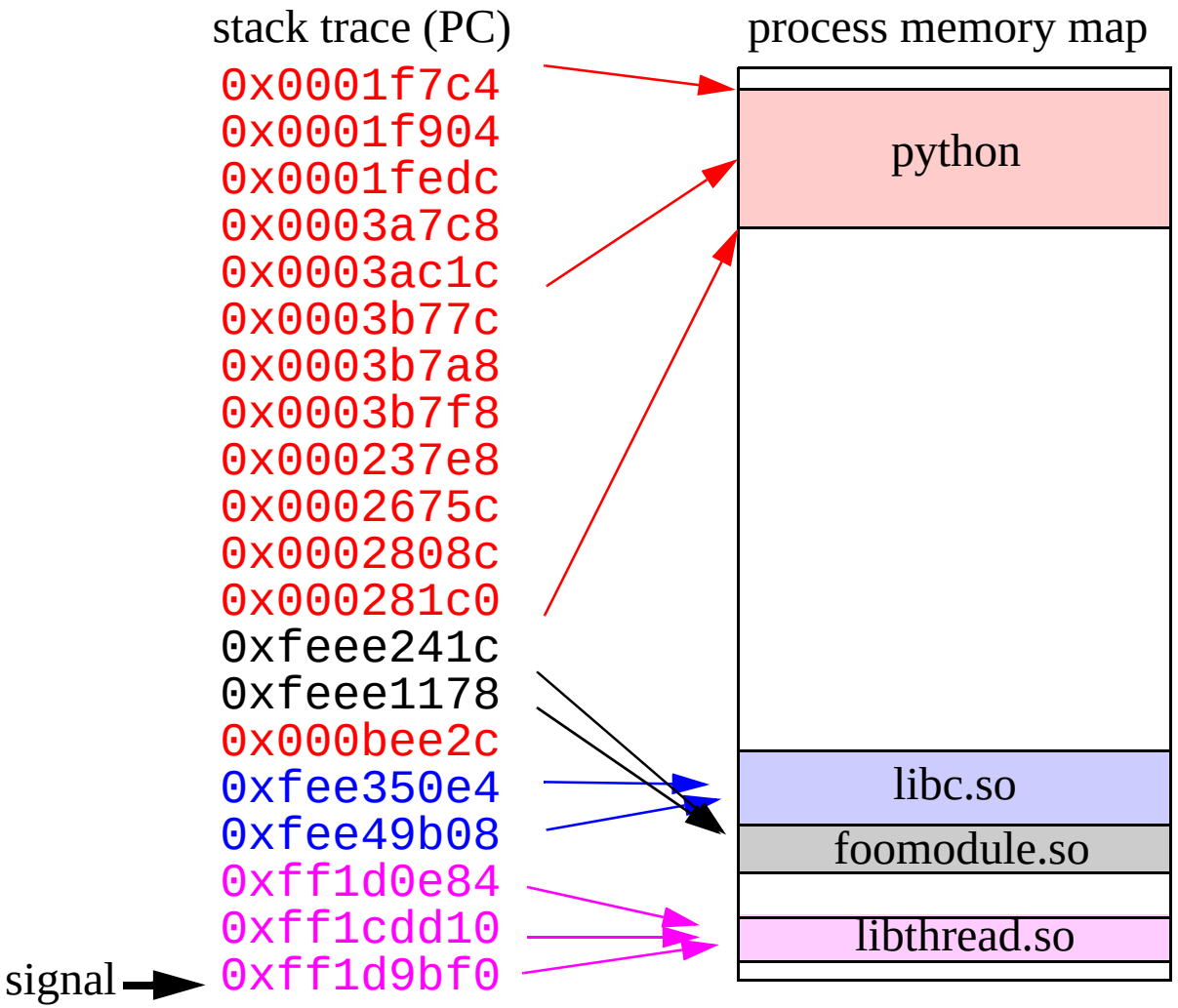
Finding Program Context



2. Read process memory map from /proc

- Get base/bounds for Python executable, all shared libraries, heap, stack, etc.

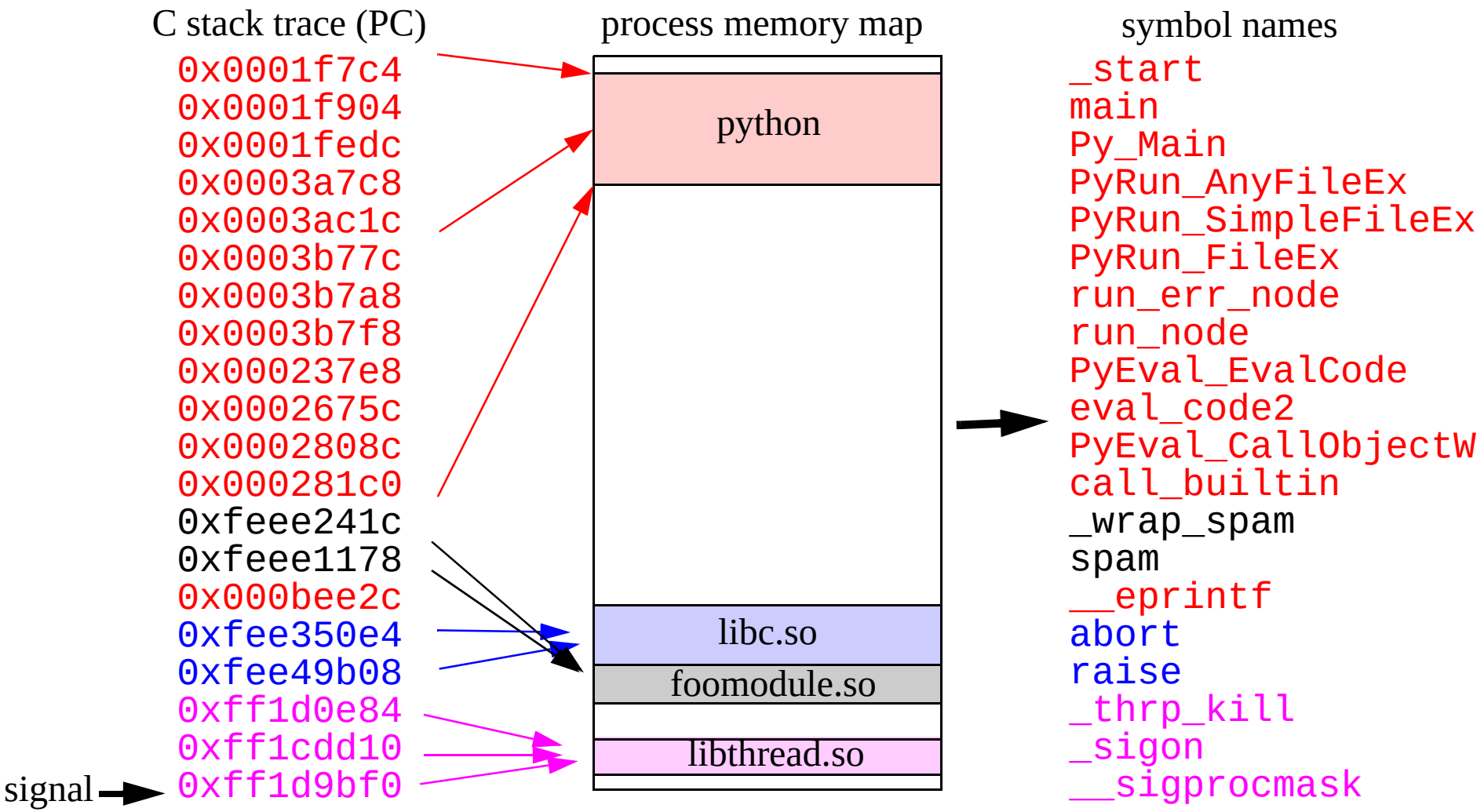
Program Context



3. Map stack trace to memory map

- Determines the module associated with each stack frame.
- Note: memory map also used to validate the stack trace.

Program Context



4. Map to symbolic names

- Read ELF symbol table from object files in memory map
- Symbols defined by a simple (name, base, size) triple.

Gathering Debugging Information

Items of interest

- Source filename
- Source line number
- Function parameters (names, values)
- Local variables (names, values).

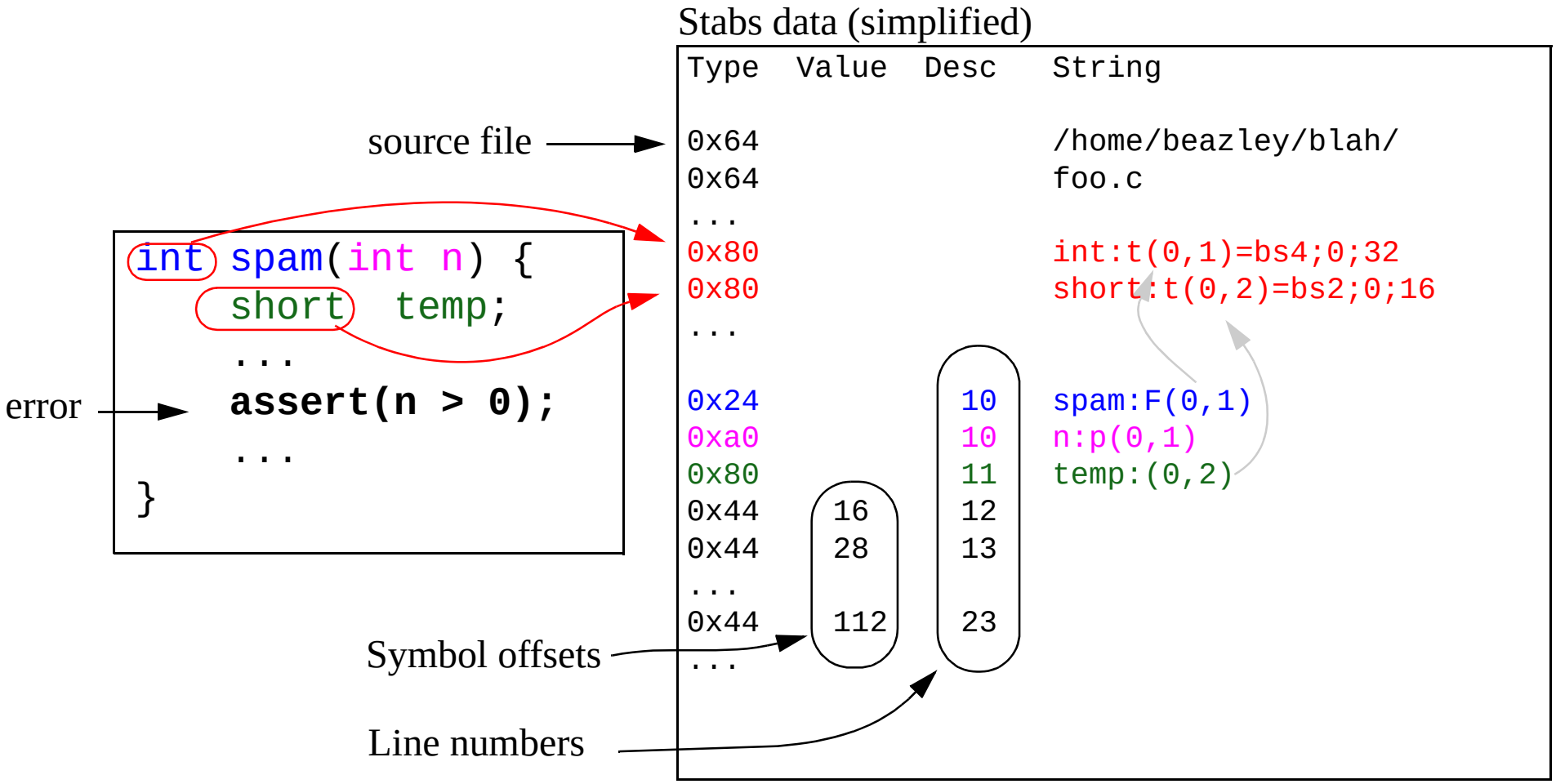
Debugging information is stored in object files

- If code compiled with -g
- However, debugging data is not loaded into memory during execution.

Collection strategy

- Load all object files found in process memory map.
- Search for debugging data for each symbol in the stack trace.

Gathering Debugging Information

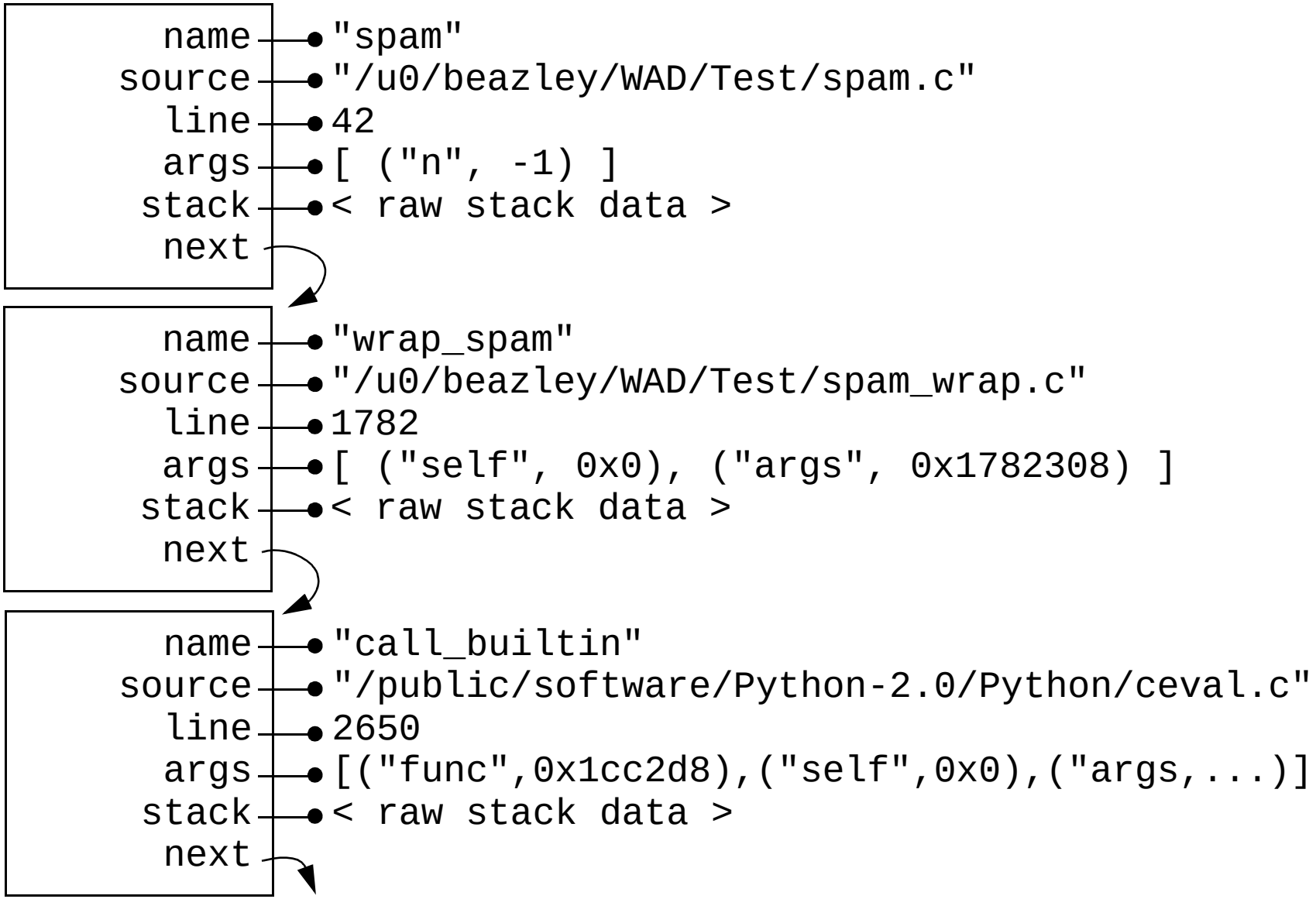


STABS

- Language neutral specification of source information.
- Includes locations, types, functions, parameters, locals, line numbers, etc.
- Decoding is a major head explosion (and that's all I will say about it).

Final Result

Get a C data structure representing program state



Returning to Python

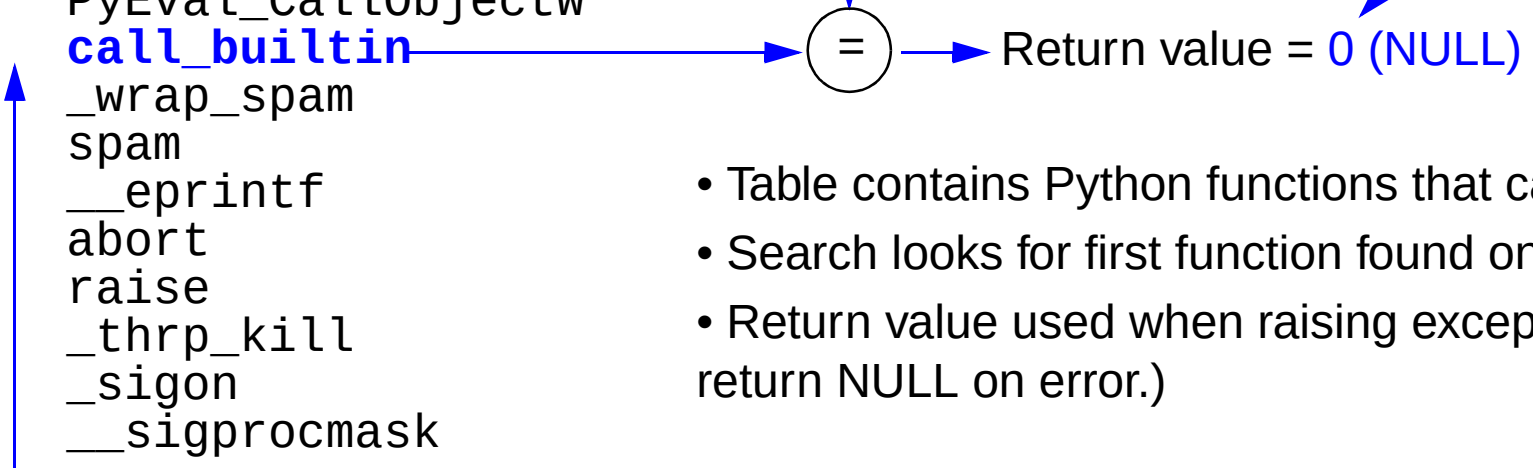
Step 1: Examine stack trace for a suitable return point

Call Stack

```
_start
main
Py_Main
PyRun_AnyFileEx
PyRun_SimpleFileEx
PyRun_FileEx
run_err_node
run_node
PyEval_EvalCode
eval_code2
PyEval_CallObjectW
call_builtin
_wrap_spam
spam
__eprintf
abort
raise
_thrp_kill
_sigon
__sigprocmask
```

Return Table

Function name	Value
call_builtin	**0**
PyObject_GetattrString	0
PyObject_SetattrString	-1
...	



- Table contains Python functions that call ext. code.
- Search looks for first function found on stack.
- Return value used when raising exceptions (e.g., return NULL on error.)

Returning to Python

Step 2: Raise an exception

- If no valid Python return function, print stack trace and exit.
- Otherwise, raise Python exception.
- `SegFault`, `AbortError`, `BusError`, `IllegalInstruction`, `Floating-PointError`

Exception value

- Is a special Python type `WadObject`
- Contains entire stack trace and all data collected.
- Is really just a wrapper around the C data structure described earlier.
- str() and repr() methods simply dump the stack trace as a string.
- Other methods provide access to raw data.

```
try:
     # some naughty extension code
except SegFault, s:
     print "Whoa!"
     print s                  # Dump a stack trace
```

Returning to Python

Step 3: Modify process context and return

- Chop off the call stack and return with an error/exception

Call Stack

```
_start
main
Py_Main
PyRun_AnyFileEx
PyRun_SimpleFileEx
PyRun_FileEx
run_err_node
run_node
PyEval_EvalCode
eval_code2
PyEval_CallObjectW
call_builtin
_wrap_spam
spam
__eprintf
abort
raise
_thrp_kill
_sigon
__sigprocmask
```


```
_start
main
Py_Main
PyRun_AnyFileEx
PyRun_SimpleFileEx
PyRun_FileEx
run_err_node
run_node
PyEval_EvalCode
eval_code2
PyEval_CallObjectW
call_builtin
```

NULL, AbortError

A "Slight" Complication

Return mechanism is similar to:

- setjmp/longjmp in C
- C++ exception handling.

However...

- Python is not instrumented or modified in any way.
- There is no corresponding `setjmp()` call.
- There is no matching `try { ... }` clause in C++.

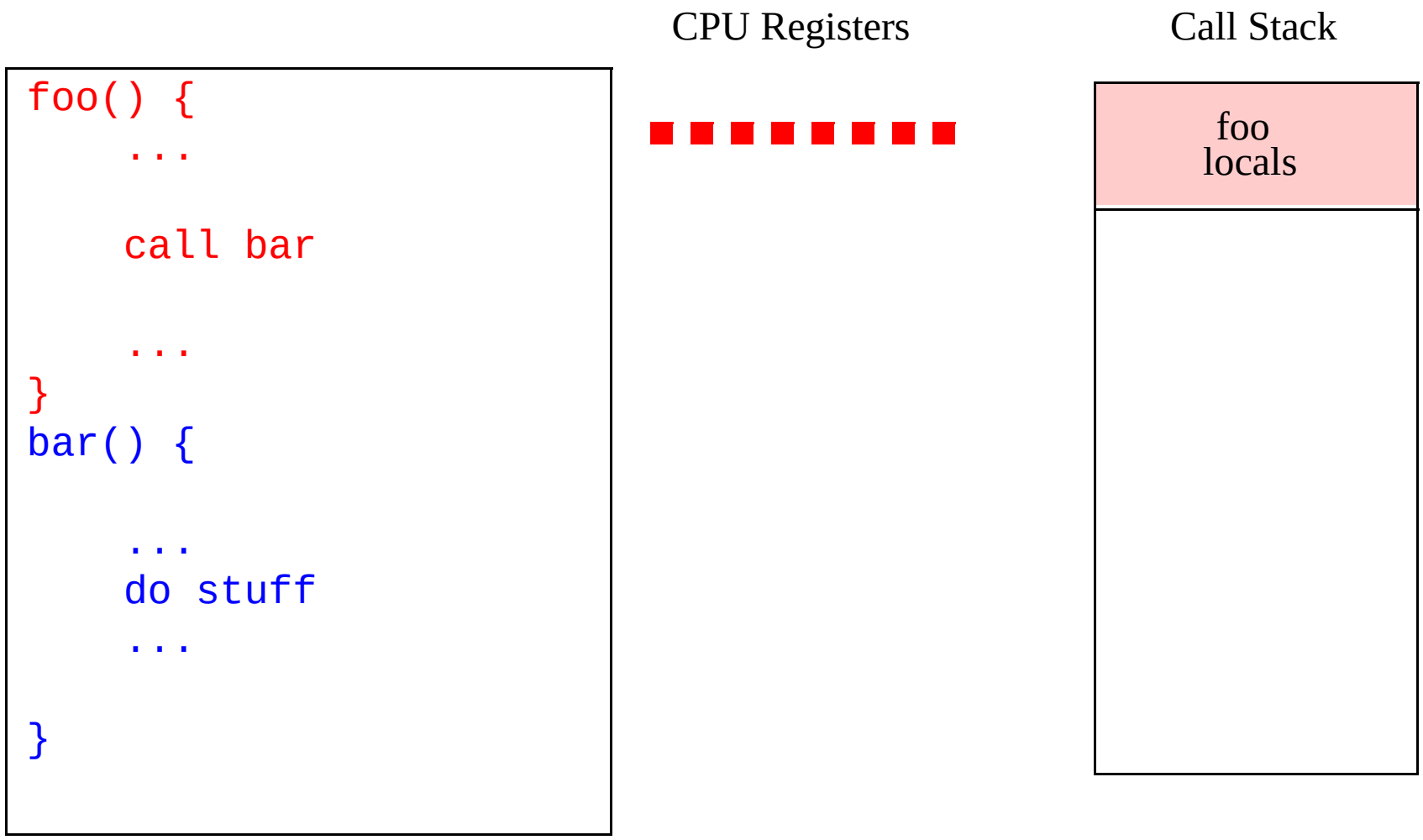
This means...

- We are returning to some "arbitrary" location in the Python executable.
- Never designed with such a non-local procedure return in mind.

This is a bit of a problem:

- Corrupted CPU registers.

The Register Save Problem



Each procedure uses CPU registers.

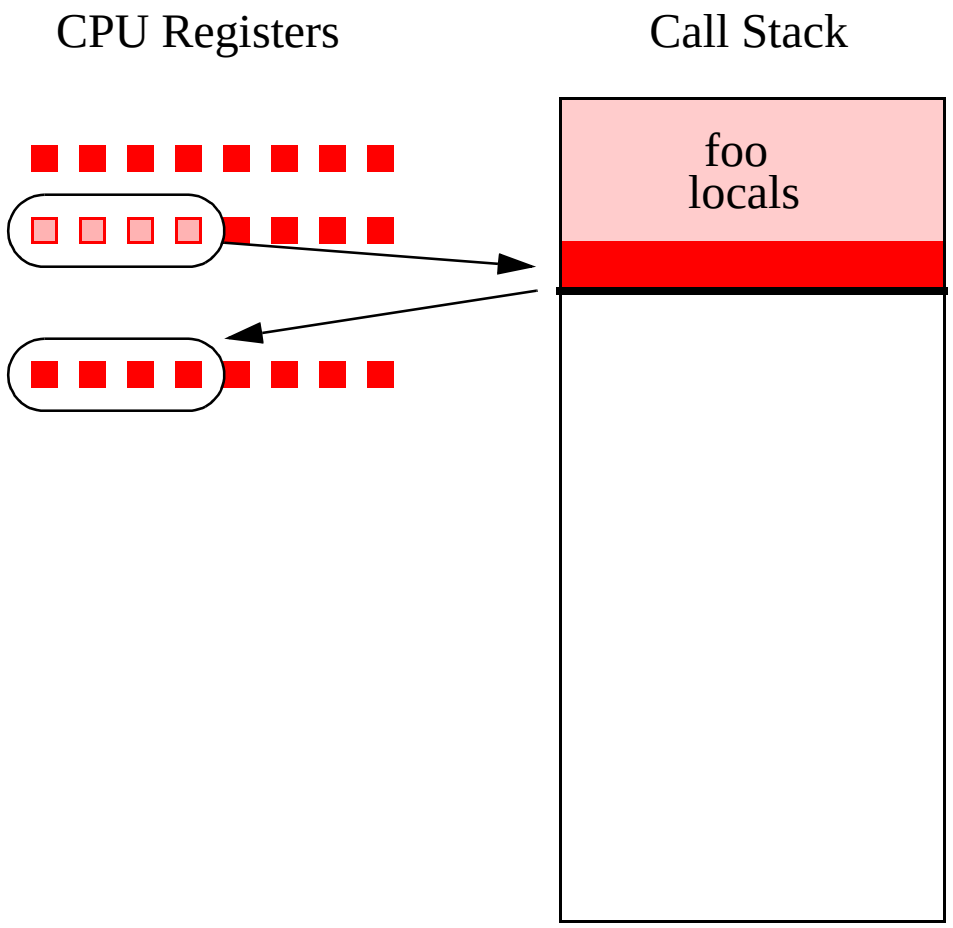
- Temporaries, local variables, memory addressing, etc.

Register Save (cont)

```
foo() {
    ...
    save_caller_regs
    call bar
    restore_caller_regs
    ...
}
bar() {

    ...
    do stuff
    ...

}
```

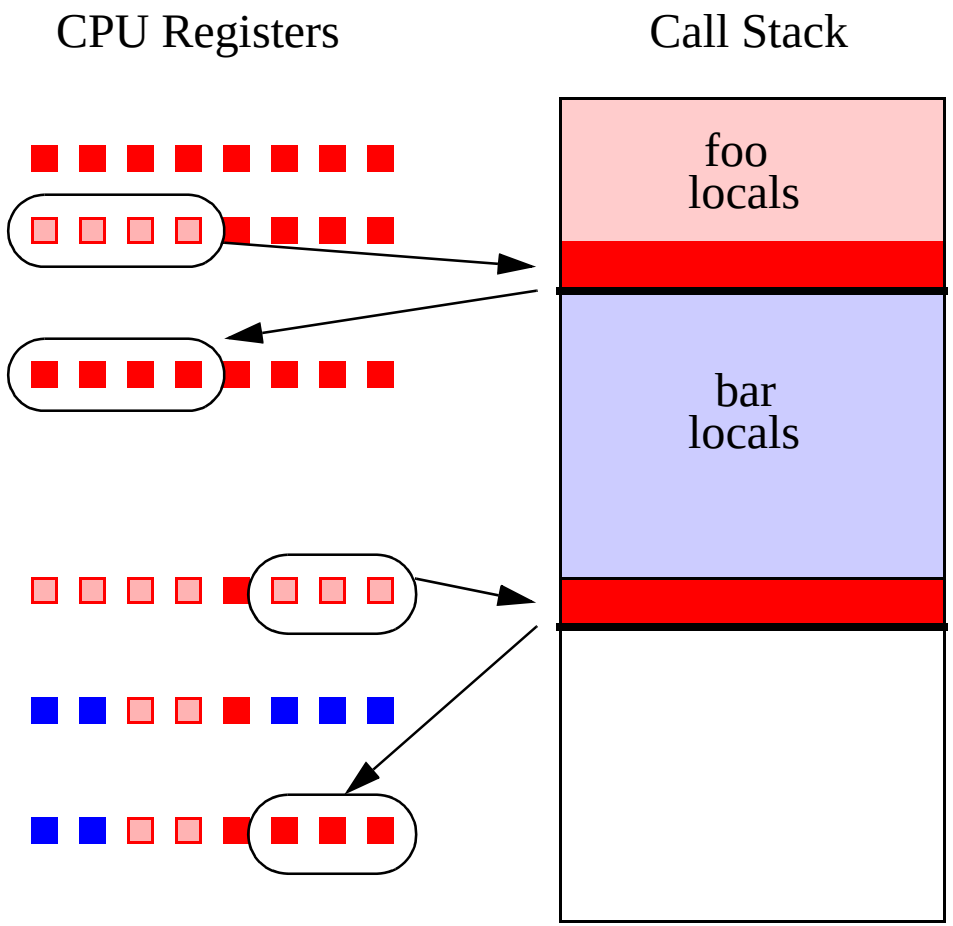


Caller-save

- Must save certain registers before calling a new procedure.
- Restore after procedure returns.

Register Save (cont)

```
foo() {
    ...
    save_caller_regs
    call bar
    restore_caller_regs
    ...
}
bar() {
    save_callee_regs
    ...
    do stuff
    ...
    restore_callee_regs
}
```

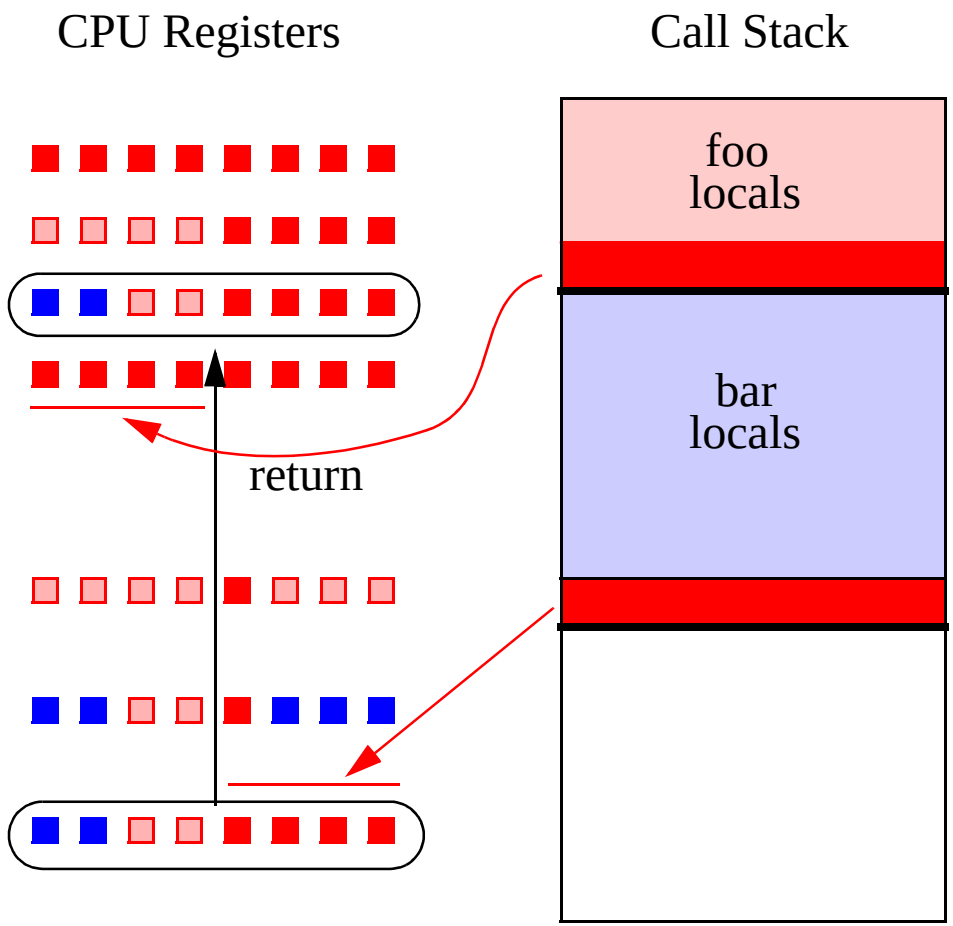


Callee-save

- Procedures save registers they plan to overwrite.
- Restore values prior to returning

Register Save (cont)

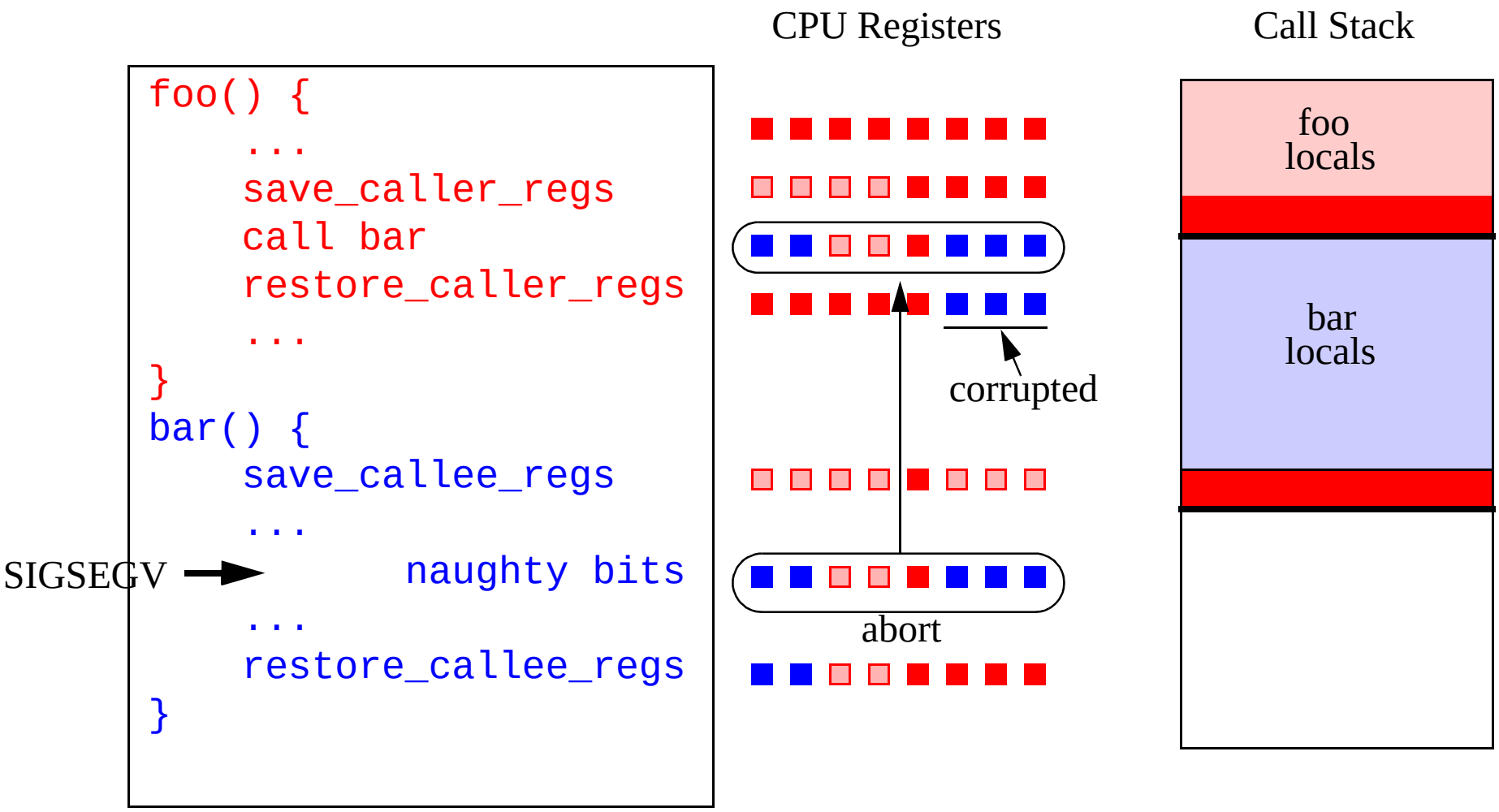
```
foo() {
    ...
    save_caller_regs
    call bar
    restore_caller_regs
    ...
}
bar() {
    save_callee_regs
    ...
    do stuff
    ...
    restore_callee_regs
}
```



Procedure return

- Callee restores registers
- Caller restores registers

Register Save (cont)



Aborted return

- Callee-saved register values are lost (never restored)
- Corrupts CPU state in caller on return (this is usually bad)

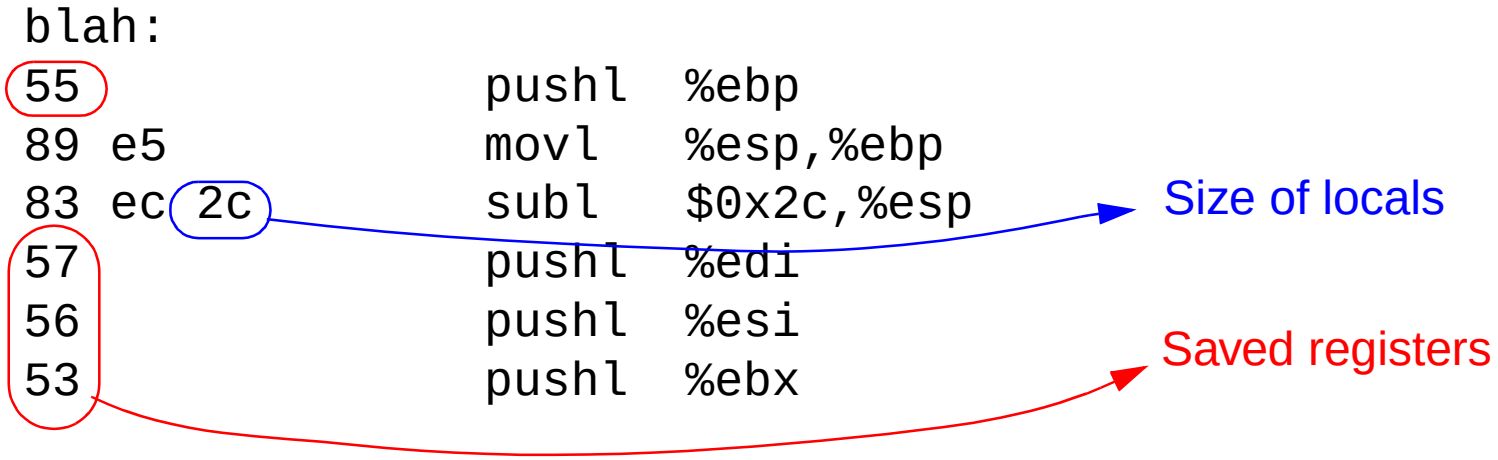
Register Restoration

Solution: SPARC

- Each procedure gets a fresh set of CPU registers (i.e., a “window”)
- To restore state: simply roll back the register windows

Solution: i386

- Manually inspect machine code of function prologues
- Figure out where callee-save registers are saved on call-stack
- Restore values while walking up the call stack.



- Only a heuristic. Might get it wrong, but the return to Python may still work.
- Not as bad as it sounds---implementation is fairly simple.

The Auto-Initialization Hack

One final bit...

- How do you get WAD to initialize itself when linked to extensions?

```
class WadInit {
public:
    WadInit() {
        wad_initialize();
    }
};
static WadInit winit;
```

- Dynamic link/loader automatically invokes C++ static constructors on import.
- Constructors are invoked before any extension code executes.

Implementation Details

Implementation

- Mostly ANSI C, some assembly, some C++
- ~1500 semicolons
- Most code related to introspection (debugging, symbol tables, etc...)
- Core is Python independent (only 166 semicolons related to Python).
- Execution is isolated (own stack and memory management).
- Does not rely upon third party libraries (e.g., libbfd).

Compatibility

- Sun Sparc Solaris
- i386 Linux (recent kernels).
- Python 1.5 and newer (class based exceptions)
- Miscellaneous compatibility issues on Linux.
- Also supports Tcl.

Limitations

Non-local return, aborted execution

- May leak memory
- No destruction of objects in C++.
- May interact poorly with C++ exceptions.
- May result in unreleased system resources (files, sockets, etc.).
- May result in deadlock (if holding locks when error occurs).

Unrecoverable errors

- Extensions that destroy or corrupt Python interpreter data.
- Stack overflow (results in corrupted call-stack).

Compiler optimization

- False reporting of debugging data, source files, and lines.
- Incorrect register recovery (-fomit-frame-pointer)

Compatibility

- Mixing threads and signals is extremely problematic.
- WAD requires fully functional signal implementation.
- Some versions of Linux, Linux+Threads do not work.

More Limitations

Debugging information

- Only simple datatypes are currently understood.
- No special C++ support (classes, name demangling, etc.)
- No understanding of structures.

Things that just don’t work

- Breakpoints
- Single-step execution.
- Restart

Related Work

Surprisingly little literature on this topic

- PyDebug.
- Programming environments for Common Lisp (FFI).
- Asynchronous exception handling (ML, Haskell)
- R^n (A mixed interpreter-compiled system for Fortran)
- Modifications to gdb for debugging Common Lisp (WCL).
- Java mixed-mode debugging (Java + JNI). ???
- Perl (sigtrap module can print perl stack trace on fatal error).

Future Directions

Better error recovery and data reporting

- Make the WAD core as generic as possible.
- Better heuristics for certain errors (corrupted call stack, corrupted heap).
- Improved collection of debugging information.

Support for more platforms

- Obviously. Maybe. Not.

Integration with Python debugger, IDEs?

- Demo.

Other languages

- Tcl, Ruby, Perl, etc. (Tcl works now).

Bizarre execution modes?

- Restarts?
- Breakpoints?
- Code patching?

Conclusions

Extension programming

- A lot of people are building extensions.
- Debugging has always been a little annoying.

Conventional wisdom

- Modify an existing debugger to understand Python.
- Why reinvent the wheel (especially debuggers)?

Why not reevaluate the situation?

- Traditional debugging model is awkward for extension programming.
- Exception handling approach is cool and fits in nicely with Python scripts.
- Simply knowing where code crashed is enough to fix a lot of bugs.
- The exception approach is also nice when distributing extensions.

Bottom line: WAD is mostly a proof of concept

- Common extension errors can be handled within Python.
- Can extend Python exception handling to compiled extensions.

More Information

http://systems.cs.uchicago.edu/wad

- This is work in progress.
- Not ready for prime time yet.
- Many related problems to work on.
- Volunteers welcome.
- I’m also looking for students.