Xen Project
DEVELOPER AND DESIGN SUMMIT

SUPERFLUIDITY

JULY 11-13, 2017
BUDAPEST, HUNGARY

uniprof: Transparent Unikernel Performance Profiling & Debugging

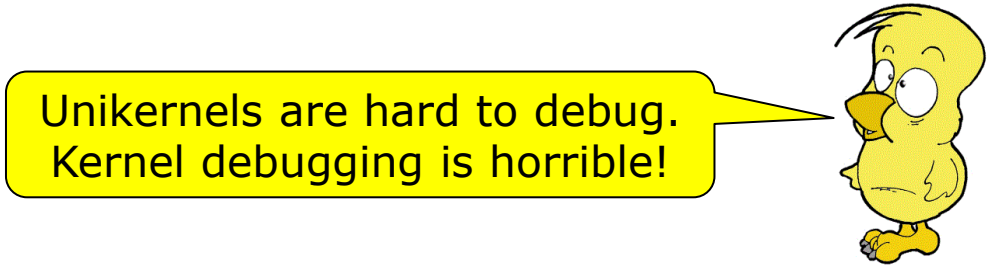
Florian Schmidt, Research Scientist, *NEC Europe Ltd.*

Unikernels?

- Faster, smaller, better!

Unikernels?

- Faster, smaller, better!
- But ever heard this?

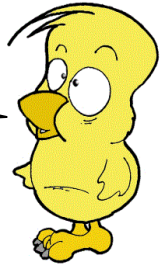


clip arts: clipproject.info

Unikernels?

- Faster, smaller, better!
- But ever heard this?

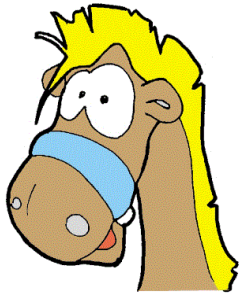
Unikernels are hard to debug.
Kernel debugging is horrible!



clip arts: clipproject.info

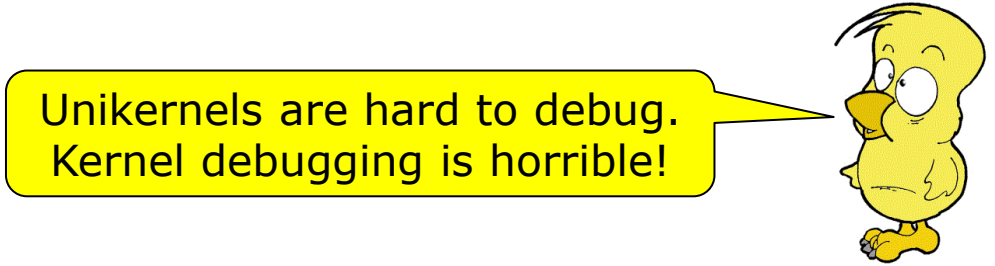
- Then you might say

But that’s not really true!
Unikernels are a single linked binary.
They have a shared address space.
You can just use gdb!



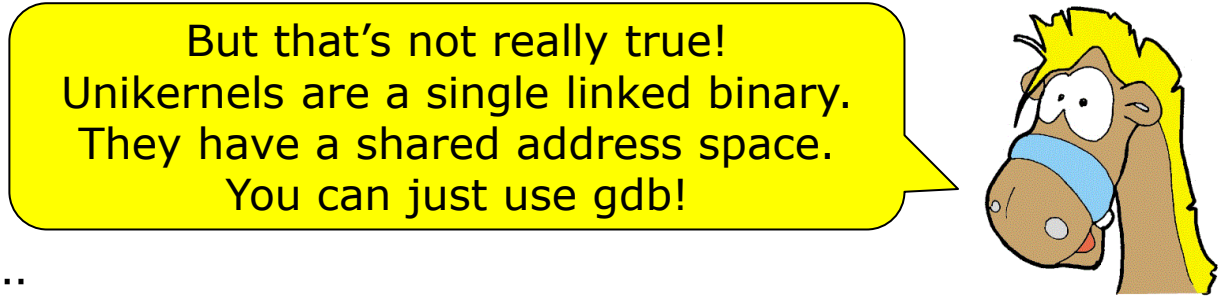
Unikernels?

- Faster, smaller, better!
- But ever heard this?



clip arts: clipproject.info

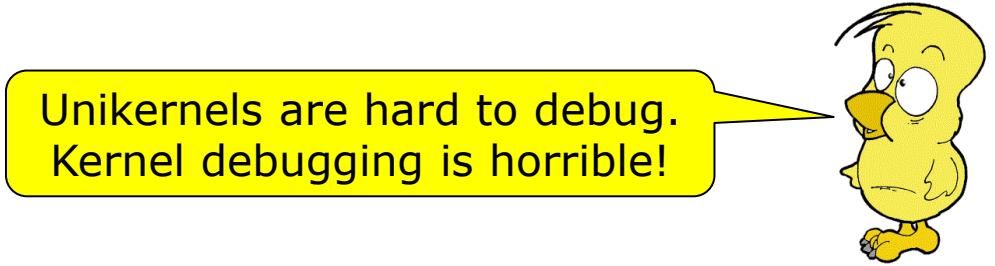
- Then you might say



- And while that is true…
- … we are admittedly lacking tools

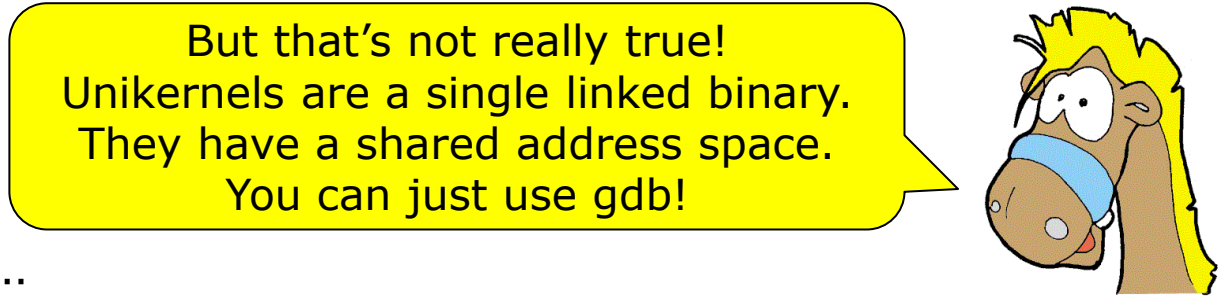
Unikernels?

- Faster, smaller, better!
- But ever heard this?



clip arts: clipproject.info

- Then you might say



- And while that is true…
- … we are admittedly lacking tools
- Such as effective profilers

Enter uniprof

Goals:

- Performance profiler
- No changes to profiled code necessary
- Minimal overhead

Enter uniprof

Goals:

- Performance profiler
- No changes to profiled code necessary
- Minimal overhead
- ➔ Useful in production environments

Enter uniprof

Goals:

- Performance profiler
- No changes to profiled code necessary
- Minimal overhead
- ➔ Useful in production environments

So, a stack profiler

- Collect stack traces at regular intervals

```
call_main+0x278
main+0x1c
schedule+0x3a
monotonic_clock+0x1a
```

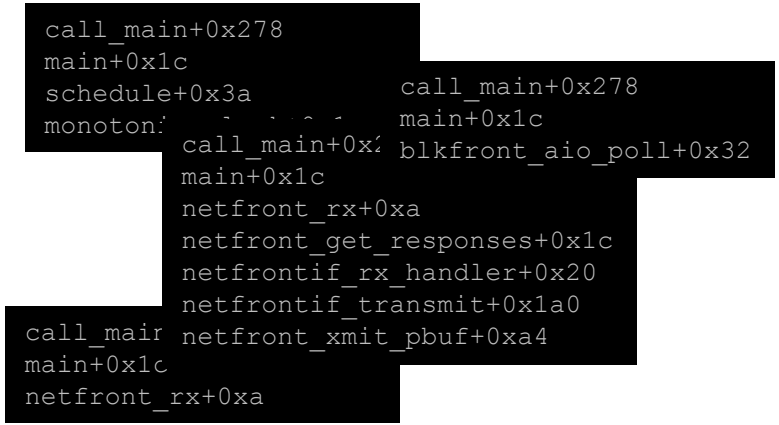
Enter uniprof

Goals:

- Performance profiler
- No changes to profiled code necessary
- Minimal overhead
- ➔ Useful in production environments

So, a stack profiler

- Collect stack traces at regular intervals
- Many of them



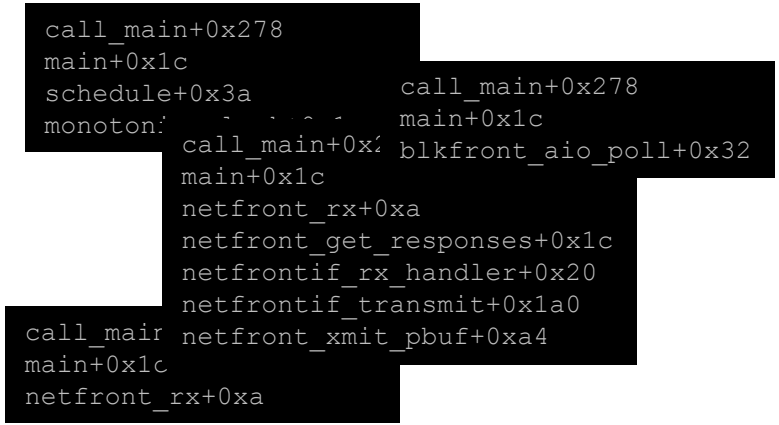
Enter uniprof

Goals:

- Performance profiler
- No changes to profiled code necessary
- Minimal overhead
- ➔ Useful in production environments

So, a stack profiler

- Collect stack traces at regular intervals
- Many of them
- Analyze which code paths show up often
  - Either because they take a long time
  - Or because they are hit often
- Point towards potential bottlenecks



xenctx

- Turns out, a stack profiler for Xen already exists
  - Well, kinda

xenctx

Turns out, a stack profiler for Xen already exists

- Well, kinda

xenctx is bundled with Xen

- Introspection tool
- Option to print call stack

```
$ xenctx -f -s <symbol table file> <DOMID>
[...]
Call Trace:
                    [<0000000000004868>] three+0x58 <--
00000000000ffea0:   [<00000000000044f2>] two+0x52
00000000000ffef0:   [<00000000000046a6>] one+0x12
00000000000fff40:   [<000000000002ff66>]
00000000000fff80:   [<0000000000012018>] call_main+0x278
```

xenctx

- Turns out, a stack profiler for Xen already exists
  - Well, kinda

- xenctx is bundled with Xen
  - Introspection tool
  - Option to print call stack

```
$ xenctx -f -s <symbol table file> <DOMID>
[...]
Call Trace:
                    [<0000000000004868>] three+0x58 <--
00000000000ffea0:   [<00000000000044f2>] two+0x52
00000000000ffef0:   [<00000000000046a6>] one+0x12
00000000000fff40:   [<000000000002ff66>]
00000000000fff80:   [<0000000000012018>] call_main+0x278
```

- So if we run this over and over, we have a stack profiler
  - Well, kinda

xenctx

Downside: xenctx is slow

- Very slow: 3ms+ per trace
- Doesn’t sound like much, but really adds up (e.g., 100 samples/s = 300ms/s)
- Can’t really blame it, not designed as a fast stack profiler

xenctx

Downside: xenctx is slow

- Very slow: 3ms+ per trace
- Doesn’t sound like much, but really adds up (e.g., 100 samples/s = 300ms/s)
- Can’t really blame it, not designed as a fast stack profiler

Performance isn’t just a nice-to-have

- We interrupt the guest all the time
- Can’t walk stack while guest is running: race conditions
- High overhead can influence results!
- Low overhead is imperative for use on production unikernels

xenctx

Downside: xenctx is slow

- Very slow: 3ms+ per trace
- Doesn’t sound like much, but really adds up (e.g., 100 samples/s = 300ms/s)
- Can’t really blame it, not designed as a fast stack profiler

Performance isn’t just a nice-to-have

- We interrupt the guest all the time
- Can’t walk stack while guest is running: race conditions
- High overhead can influence results!
- Low overhead is imperative for use on production unikernels

First question: extend xenctx or write something from scratch?

- Spoiler: look at the talk title
- More insight when I come to the evaluation

What do we need?

What do we need?

Registers (for FP, IP)

- This is pretty easy: `getvcpucontext()` hypercall

What do we need?

Registers (for FP, IP)

- This is pretty easy: `getvcpucontext()` hypercall

Access to stack memory (to read return addresses and next FPs)

- This is the complicated step
- We need to do address resolution

What do we need?

Registers (for FP, IP)

- This is pretty easy: `getvcpucontext()` hypercall

Access to stack memory (to read return addresses and next FPs)

- This is the complicated step
- We need to do address resolution
  - Memory introspection requires mapping memory over
  - We're looking at (uni)kernel code
  - But there's still a virtual → (guest) physical resolution

What do we need?

Registers (for FP, IP)

- This is pretty easy: `getvcpucontext()` hypercall

Access to stack memory (to read return addresses and next FPs)

- This is the complicated step
- We need to do address resolution
  - Memory introspection requires mapping memory over
  - We're looking at (uni)kernel code
  - But there's still a virtual → (guest) physical resolution
  - Even in guest is PVH, can't benefit from it, because we're looking in from outside
  - So we need to manually walk page tables

What do we need?

Registers (for FP, IP)

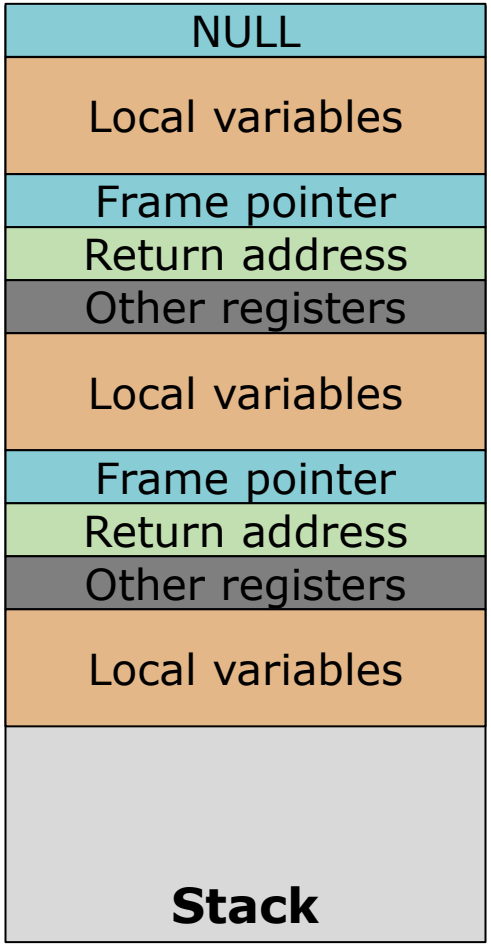
- This is pretty easy: `getvcpucontext()` hypercall

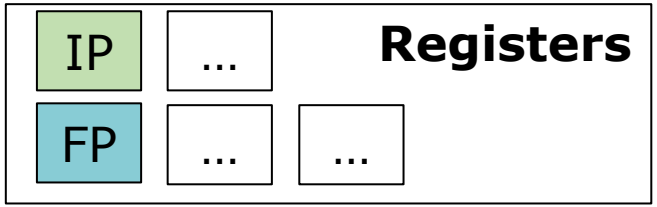
Access to stack memory (to read return addresses and next FPs)

- This is the complicated step
- We need to do address resolution
  - Memory introspection requires mapping memory over
  - We're looking at (uni)kernel code
  - But there's still a virtual → (guest) physical resolution
  - Even in guest is PVH, can't benefit from it, because we're looking in from outside
  - So we need to manually walk page tables

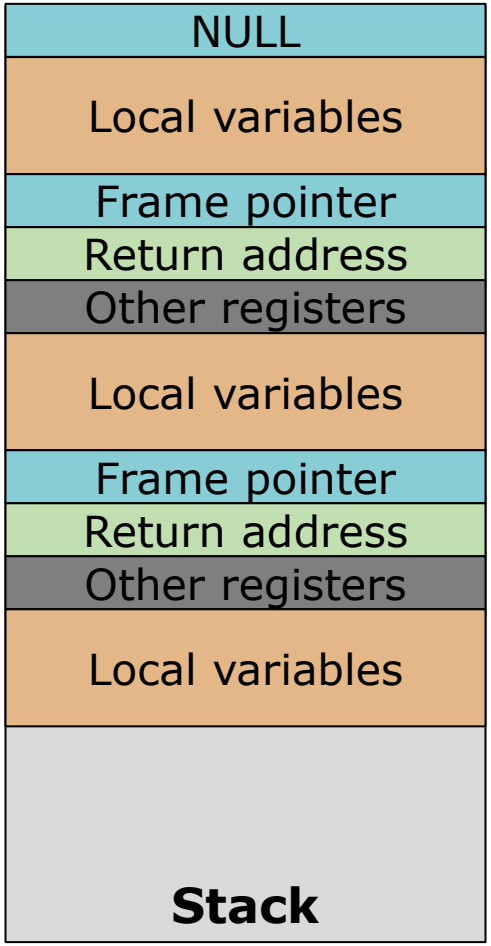
Symbol table (to resolve function names)

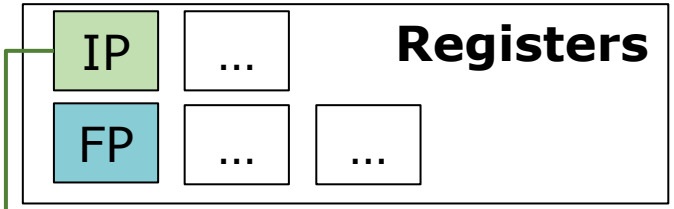
- Thankfully, this is easy again: extract symbols from ELF with `nm`





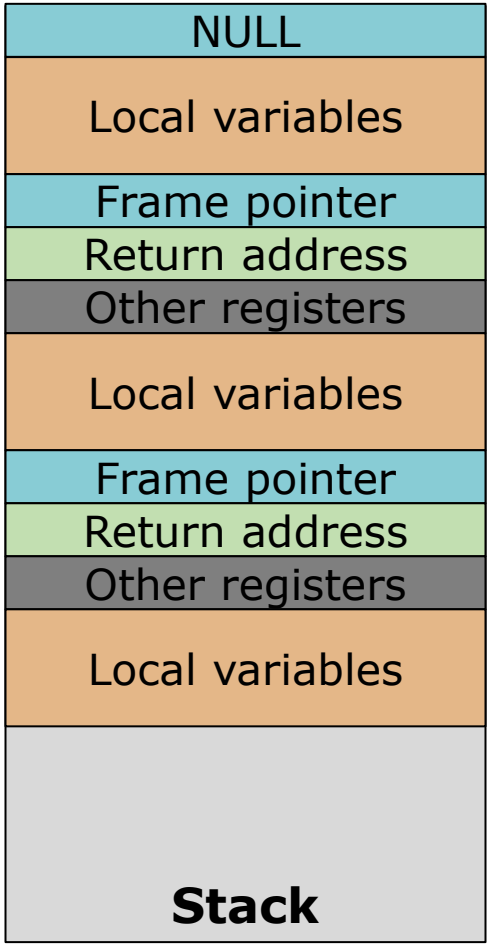

```
Stack trace:
```

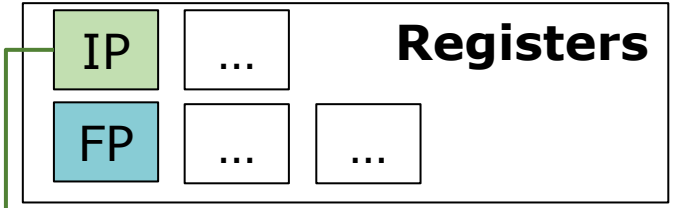


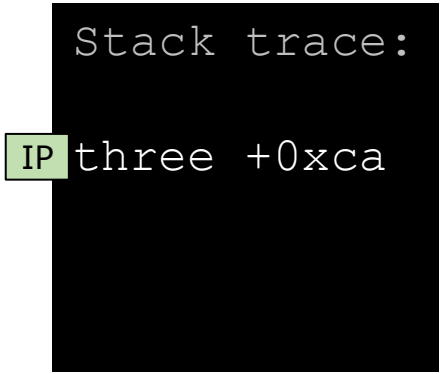



```
function three() {
    […]
}
```

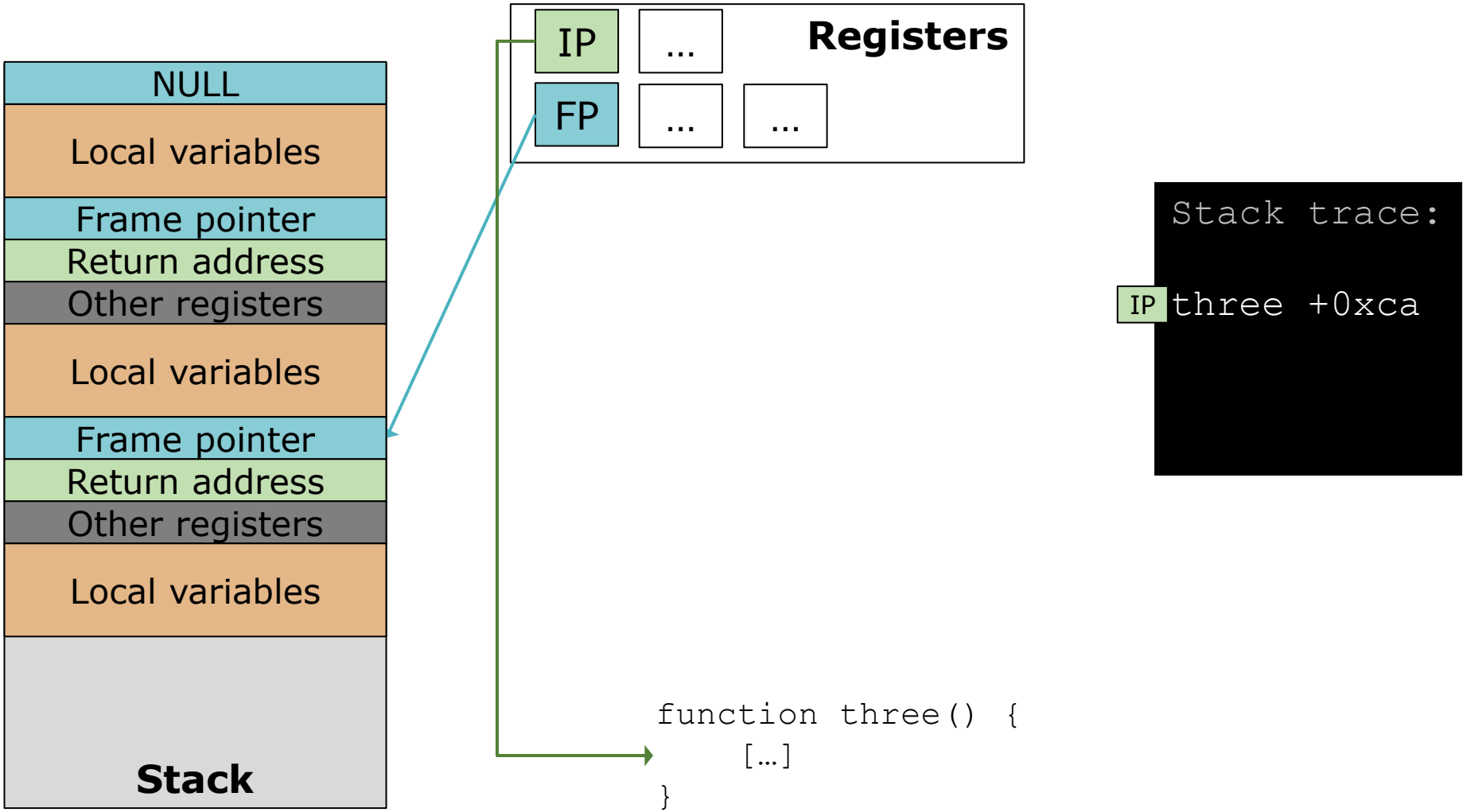
```
Stack trace:
```

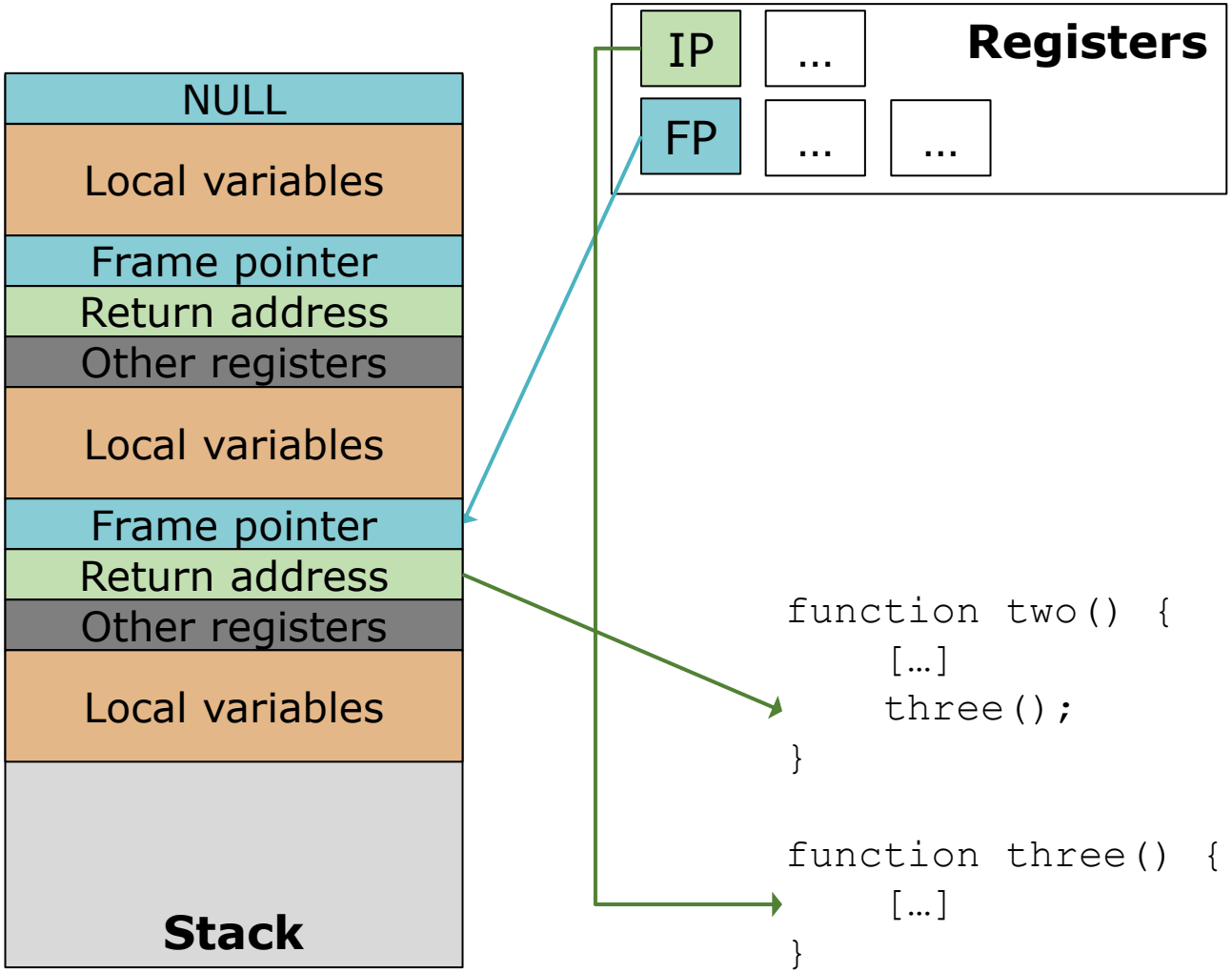


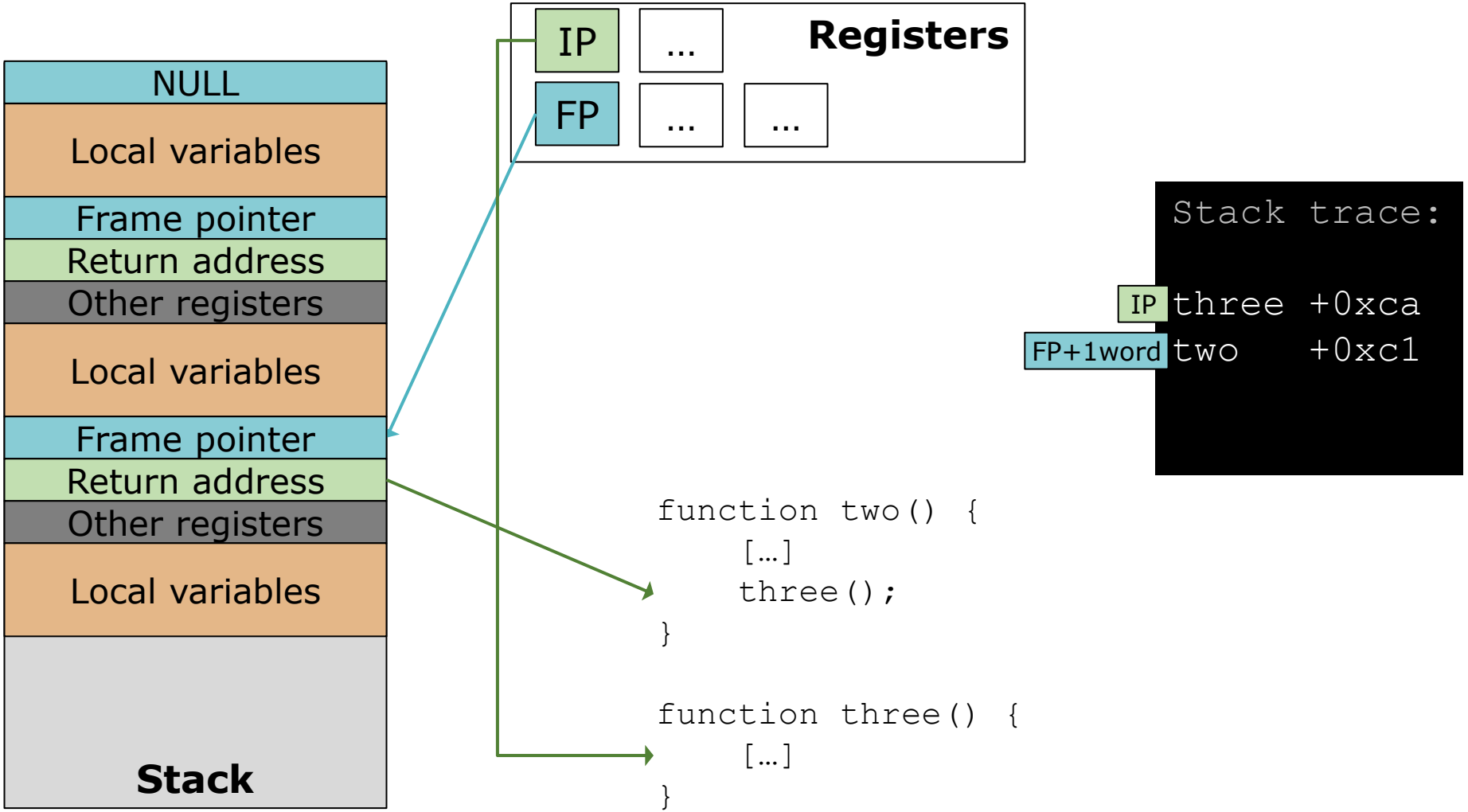


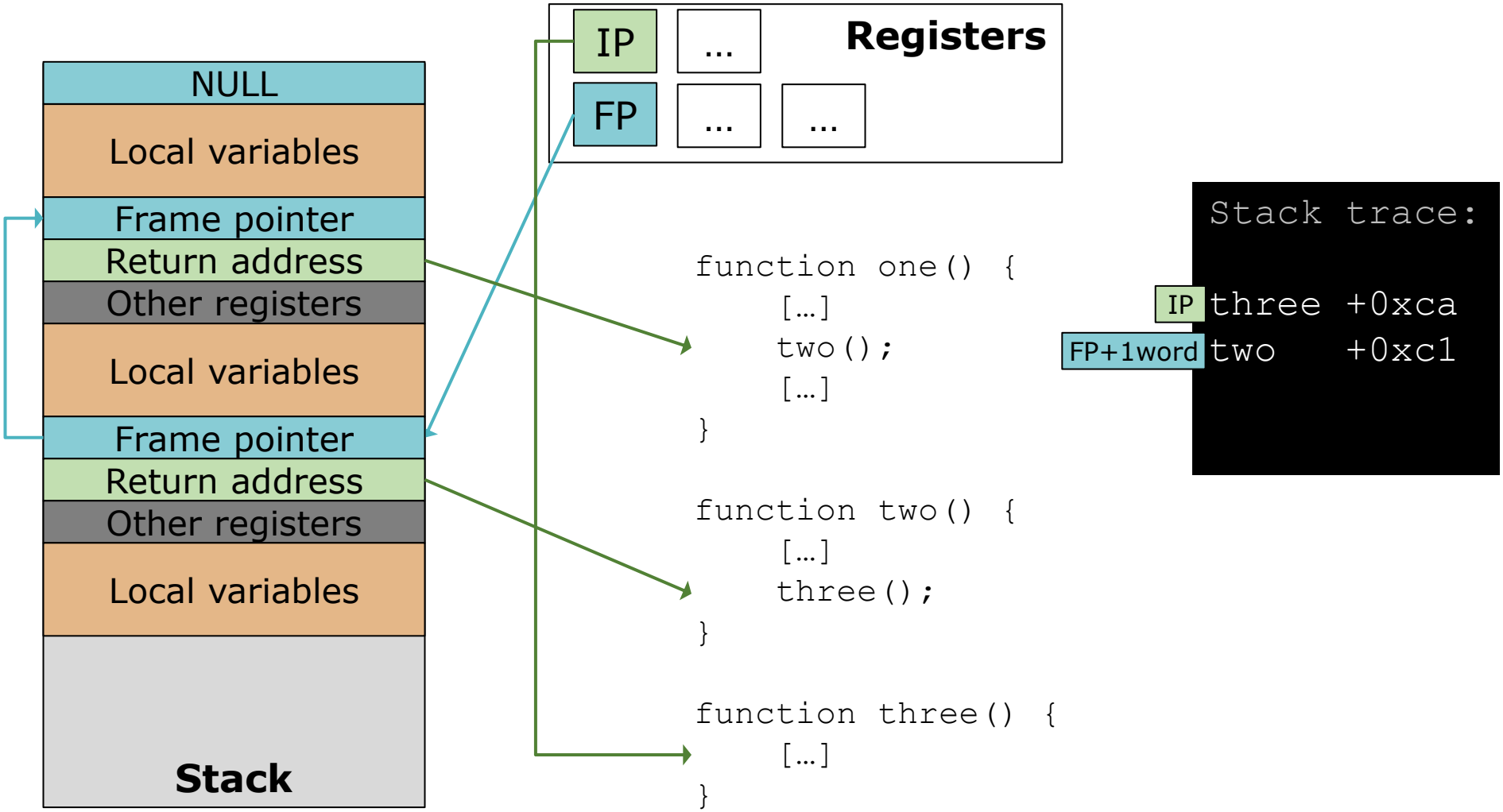


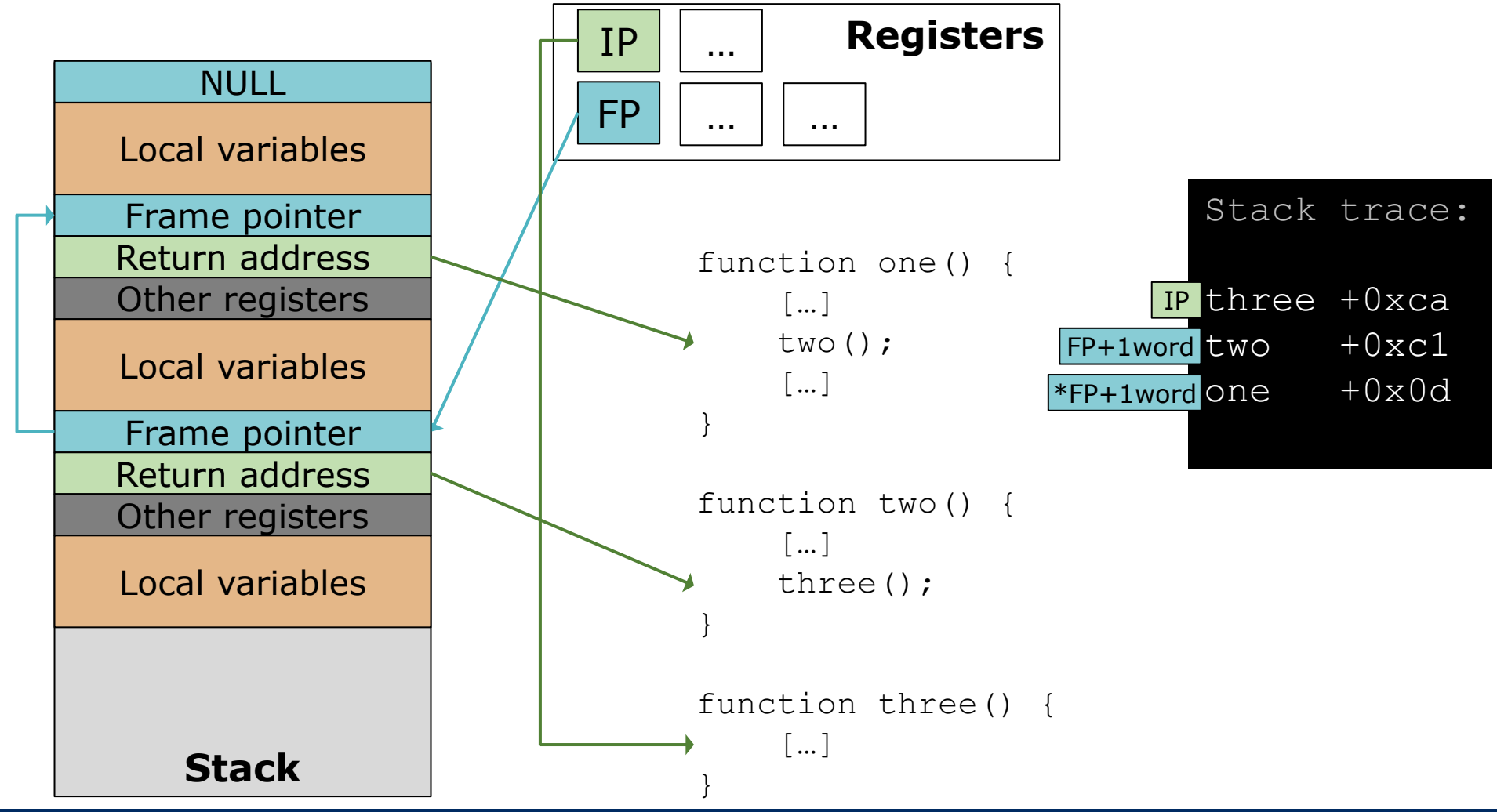

```
function three() {
    […]
}
```

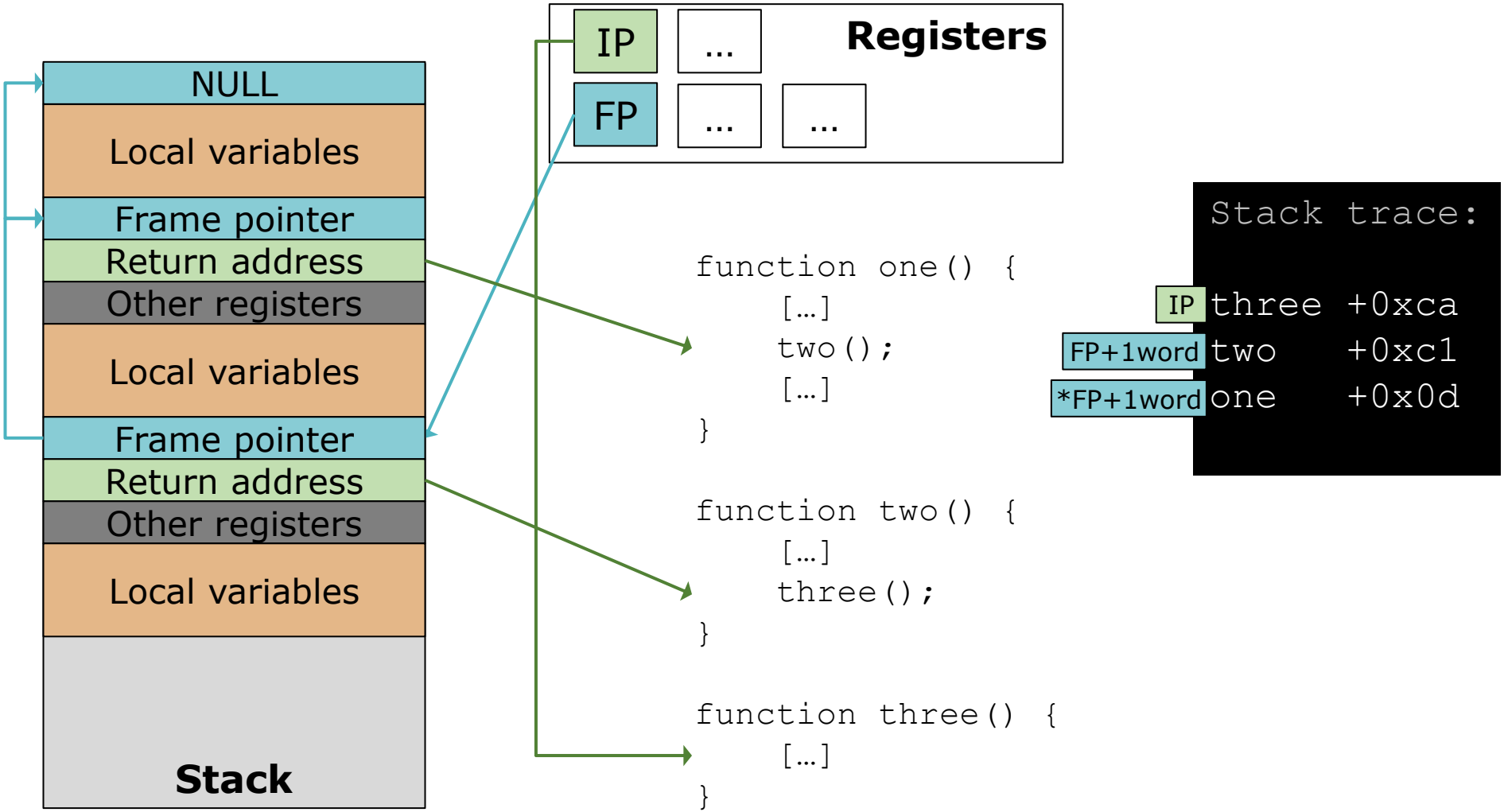


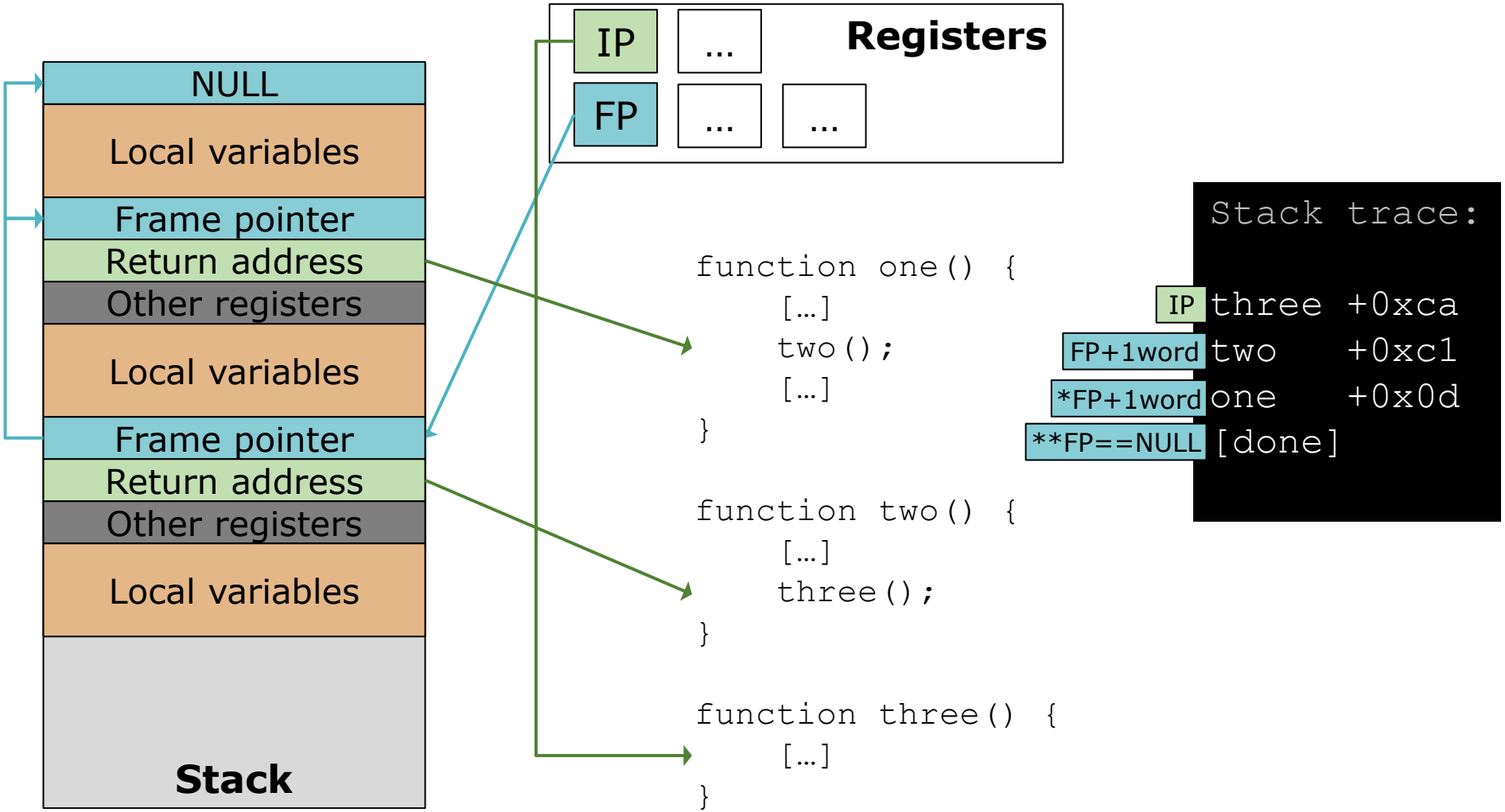




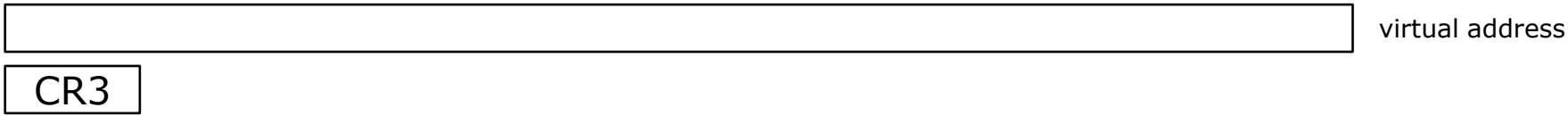




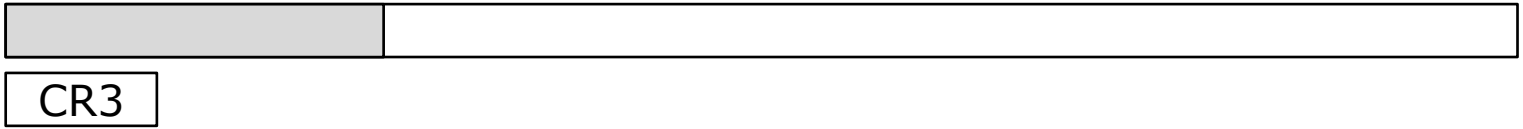




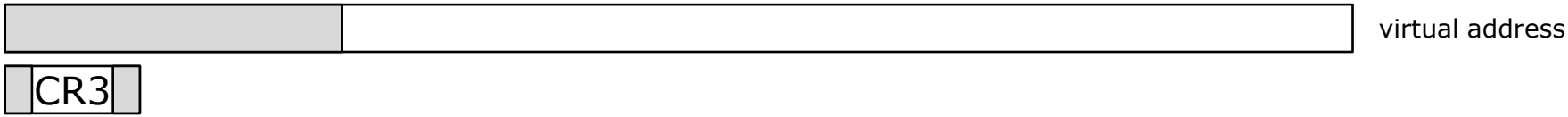
Walking the page tables (x86-64)



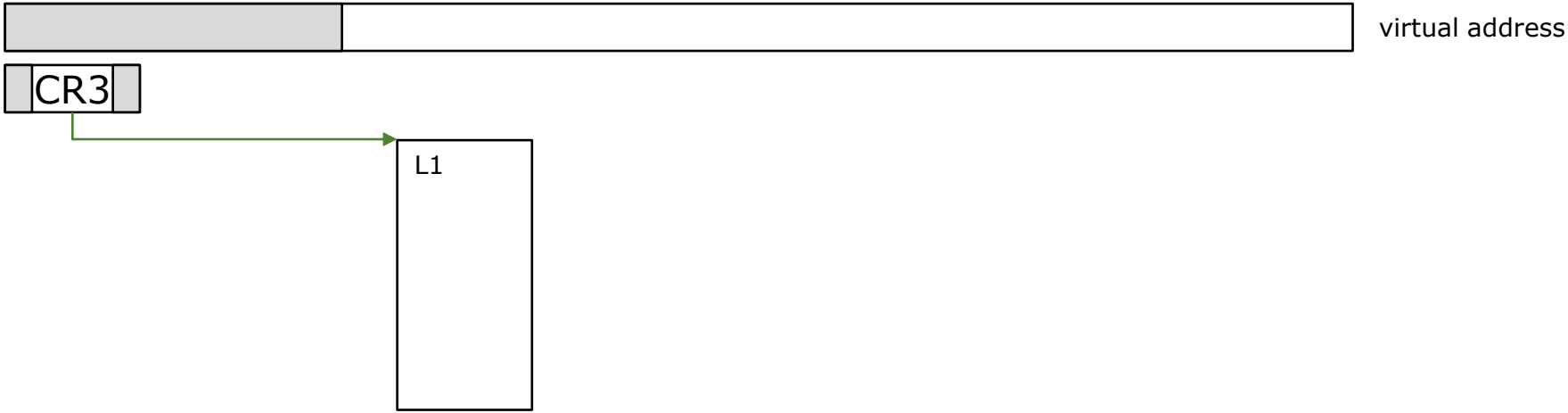
Walking the page tables (x86-64)



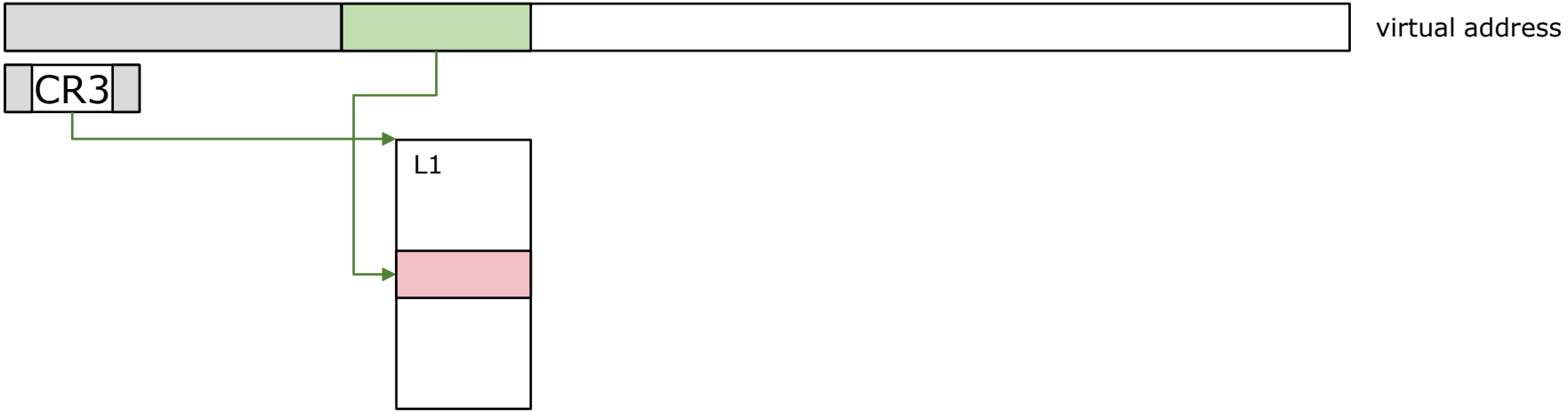
Walking the page tables (x86-64)



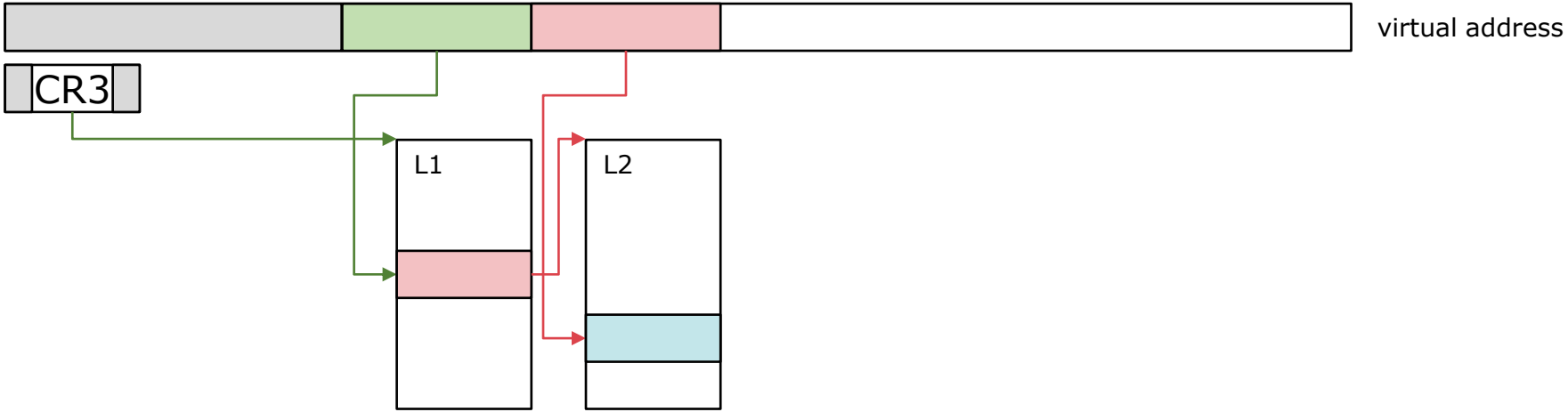
Walking the page tables (x86-64)



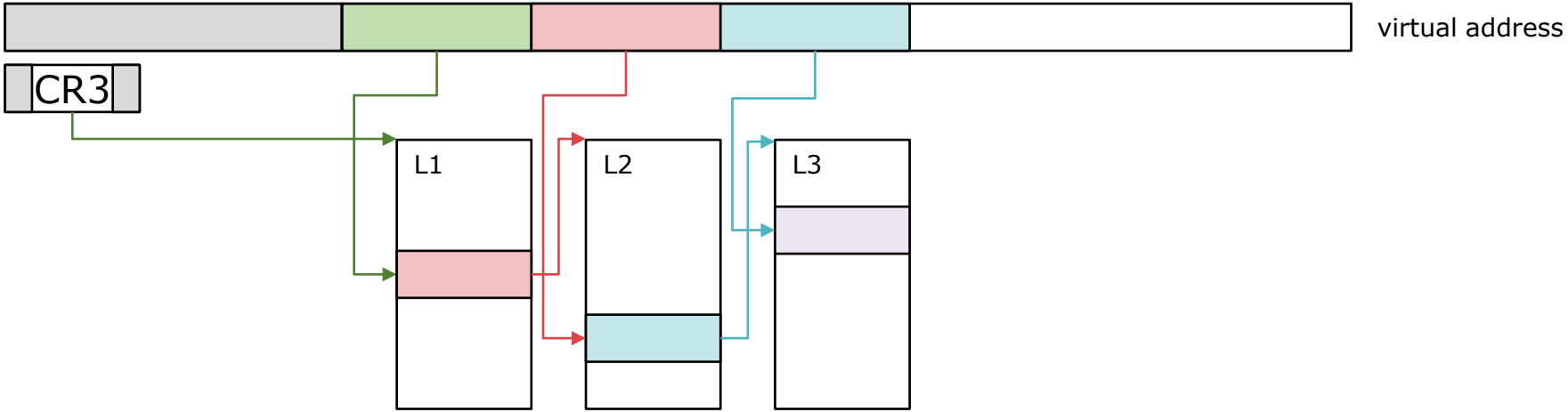
Walking the page tables (x86-64)



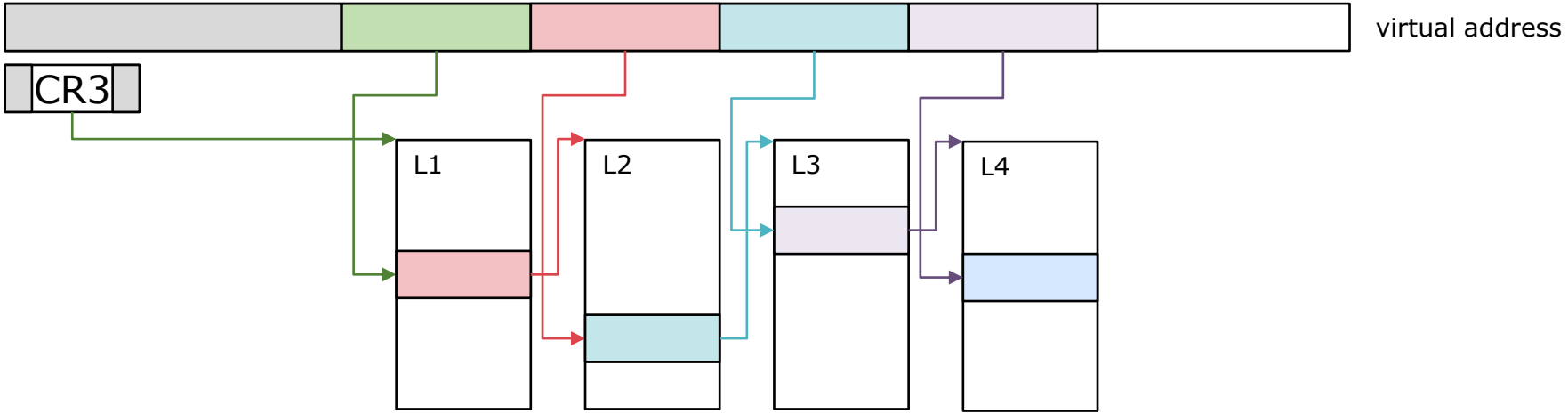
Walking the page tables (x86-64)



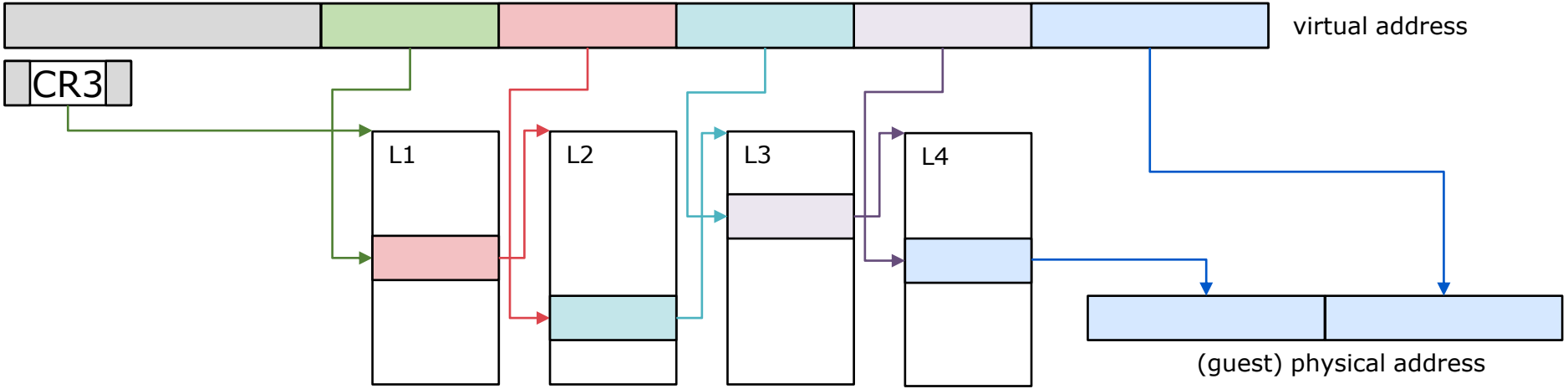
Walking the page tables (x86-64)



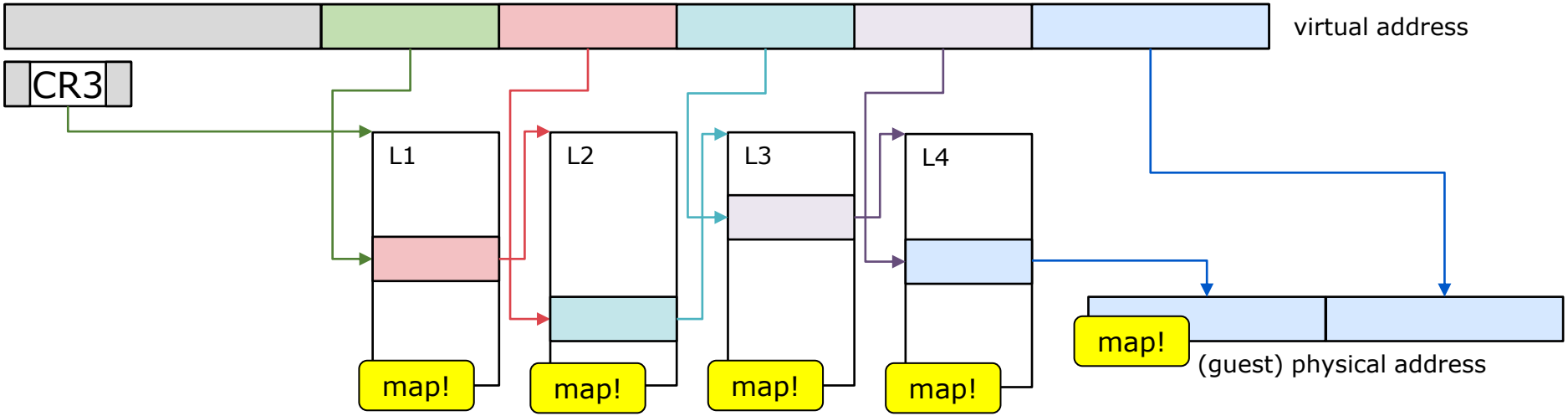
Walking the page tables (x86-64)



Walking the page tables (x86-64)



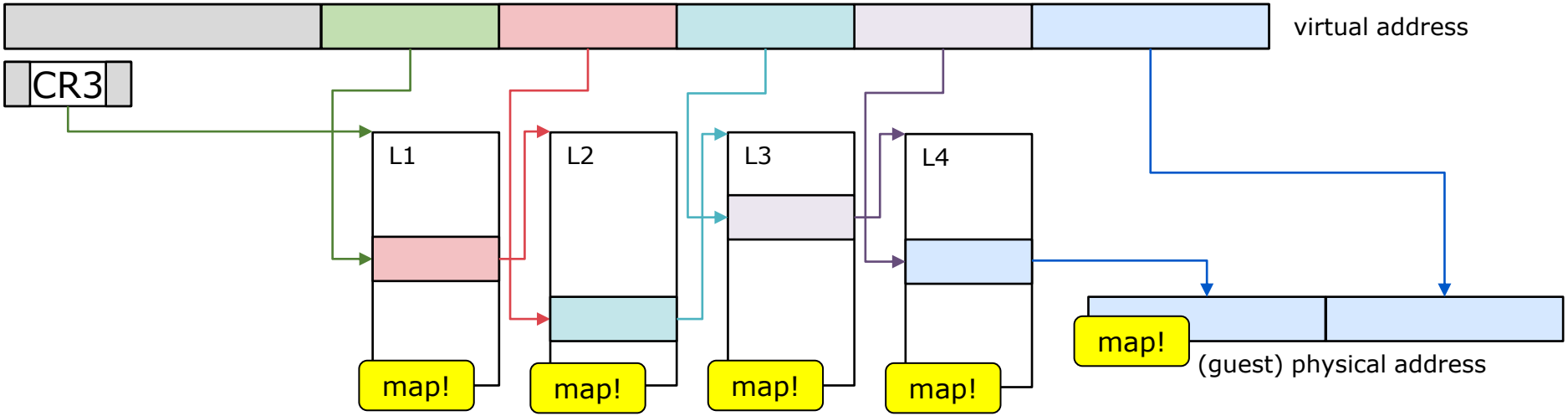
Walking the page tables (x86-64)



So many maps:

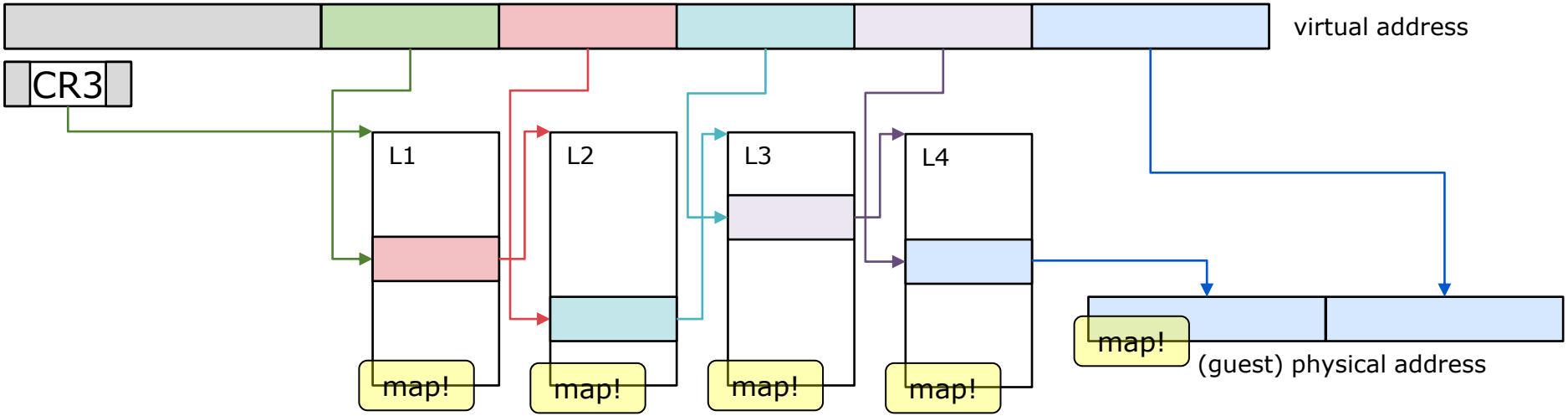
- 5 per entry * stack depth

Walking the page tables (x86-64)



- So many maps:
  - 5 per entry * stack depth
- Then again, page table locations don’t change…
  - Neither do stack locations (exception: lots of thread spawning)
  - Effective caching

Walking the page tables (x86-64)



- So many maps:
  - 5 per entry * stack depth
- Then again, page table locations don’t change…
  - Neither do stack locations (exception: lots of thread spawning)
  - Effective caching

Create Symbol Table

- Stack only contains addresses
- Symbol resolution necessary

Create Symbol Table

Stack only contains addresses

Symbol resolution necessary

Trivial

- Virtual addresses mapped 1:1 into unikernel address space
- `nm` is your friend

```
$ nm -n <ELF> > symtab
$ head symtab
0000000000000000 T _start
0000000000000000 T _text
0000000000000008 a RSP_OFFSET
0000000000000017 t stack_start
00000000000000fc a KERNEL_CS_MASK
0000000000001000 t shared_info
0000000000002000 t hypercall_page
0000000000003000 t error_entry
000000000000304f t error_call_handler
0000000000003069 t hypervisor_callback
```

Create Symbol Table

Stack only contains addresses

Symbol resolution necessary

Trivial

- Virtual addresses mapped 1:1 into unikernel address space
- `nm` is your friend

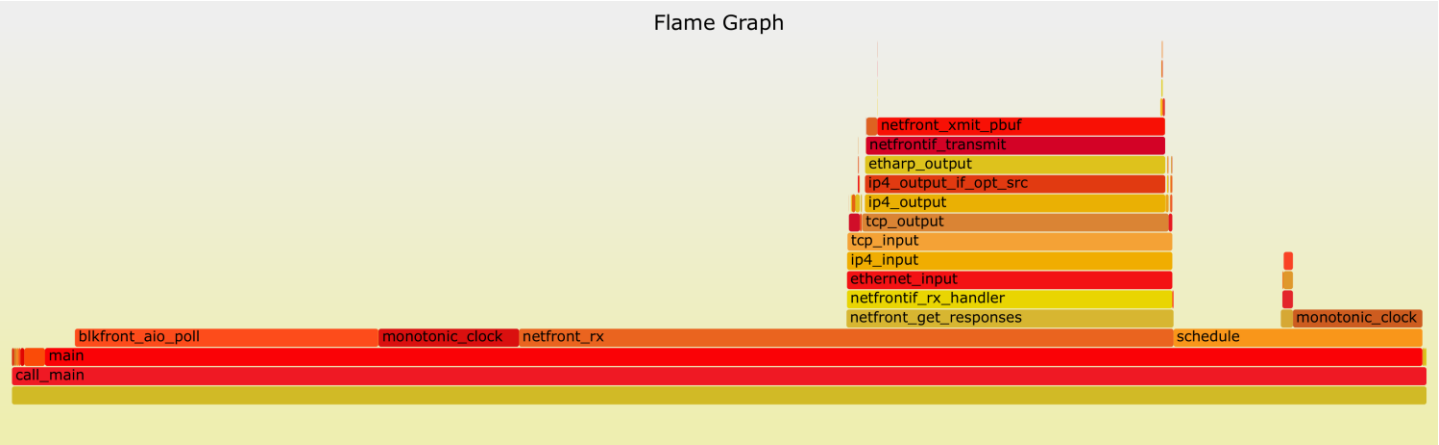
Needs unstripped binary

- You're welcome to strip it afterwards

```
$ nm -n <ELF> > symtab
$ head symtab
0000000000000000 T _start
0000000000000000 T _text
0000000000000008 a RSP_OFFSET
0000000000000017 t stack_start
00000000000000fc a KERNEL_CS_MASK
0000000000001000 t shared_info
0000000000002000 t hypercall_page
0000000000003000 t error_entry
000000000000304f t error_call_handler
0000000000003069 t hypervisor_callback
```

What do we get?

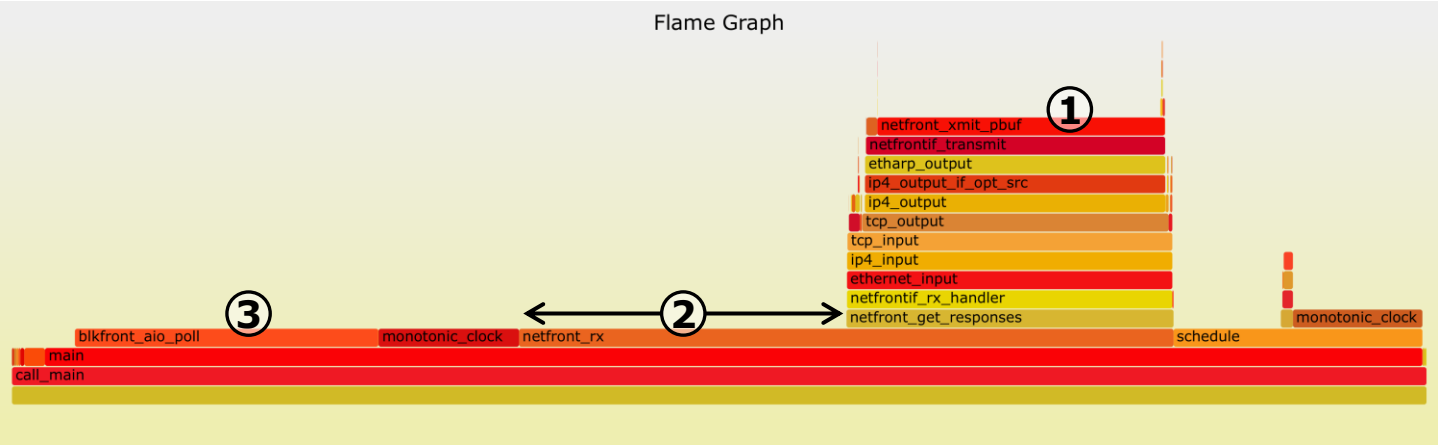
What do we get? Flamegraphs!



https://github.com/brendangregg/flamegraph

- Y Axis: call trace
  - Bottom: main function, each layer: one call depth
- X Axis: relative run time
  - Call paths are aggregated, no same call path twice in graph

What do we get? Flamegraphs!

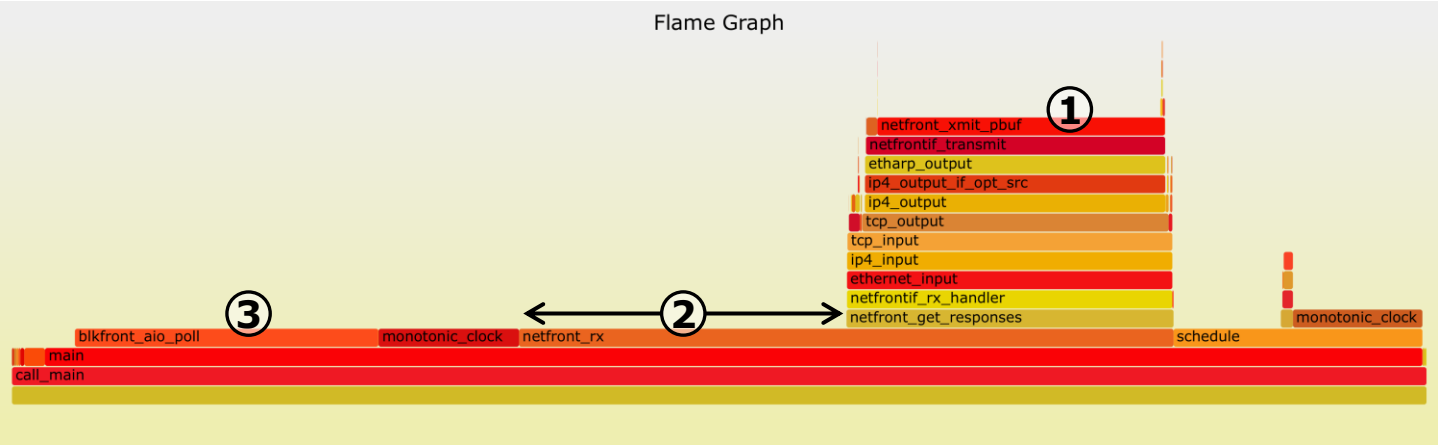


https://github.com/brendangregg/flamegraph

- Y Axis: call trace
  - Bottom: main function, each layer: one call depth
- X Axis: relative run time
  - Call paths are aggregated, no same call path twice in graph

- In this example: netfront functions “heavy hitters”
  - ① netfront_xmit_pbuf
  - ② netfront_rx
  - ③ but also blkfront_aio_poll

What do we get? Flamegraphs!



https://github.com/brendangregg/flamegraph

- Y Axis: call trace
  - Bottom: main function, each layer: one call depth
- X Axis: relative run time
  - Call paths are aggregated, no same call path twice in graph
- In this example: netfront functions “heavy hitters”
  - ① netfront_xmit_pbuf
  - ② netfront_rx
  - ③ but also blkfront_aio_poll

Yep, it’s a MiniOS* doing network communication

*with lwip for TCP/IP

Try 1: improving xenctx performance

- xenctx translates and maps memory addresses every stack walk
  - Huge overhead
  - Solution: cache mapped memory and virtual→machine translations

Try 1: improving xenctx performance

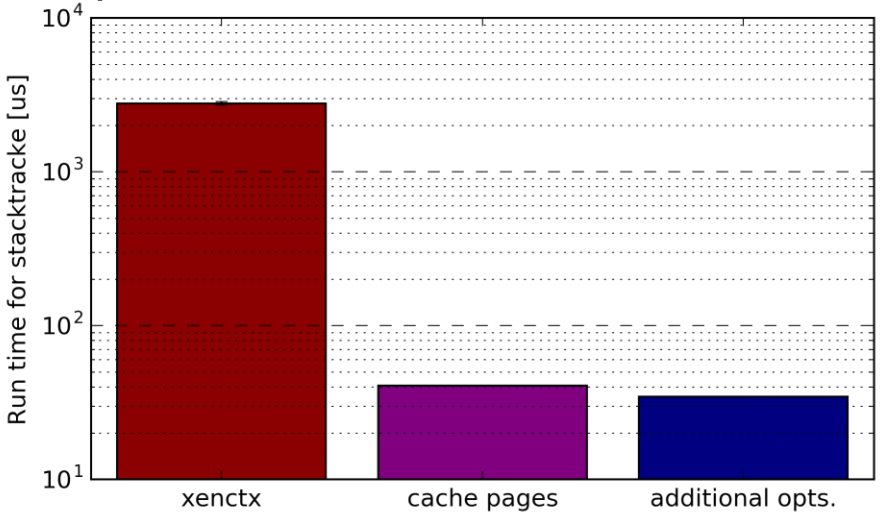
- xenctx translates and maps memory addresses every stack walk
  - Huge overhead
  - Solution: cache mapped memory and virtual→machine translations
- xenctx resolves symbols via linear search
  - Solution: use binary search

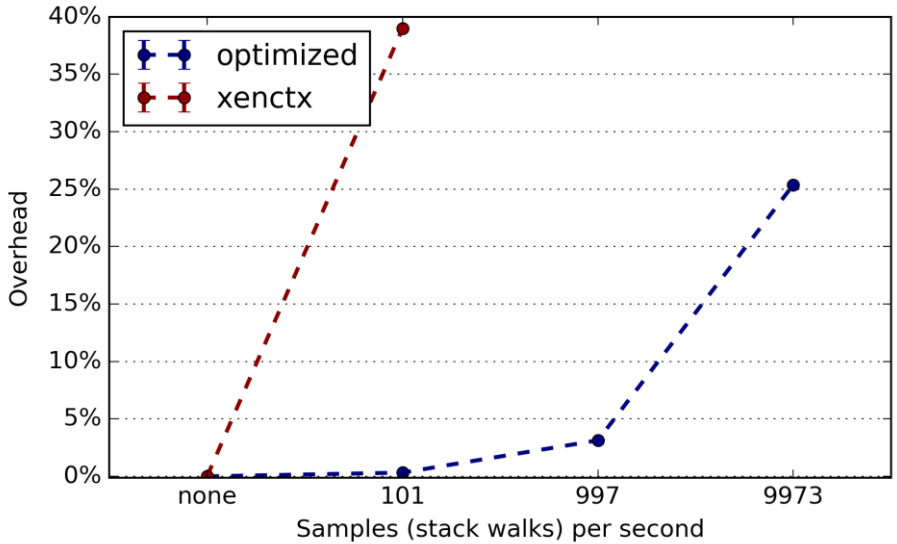
Try 1: improving xenctx performance

- xenctx translates and maps memory addresses every stack walk
  - Huge overhead
  - Solution: cache mapped memory and virtual→machine translations
- xenctx resolves symbols via linear search
  - Solution: use binary search
  - (Or, even better, do resolutions offline after tracing)

Try 1: improving xenctx performance

- xenctx translates and maps memory addresses every stack walk
  - Huge overhead
  - Solution: cache mapped memory and virtual→machine translations
- xenctx resolves symbols via linear search
  - Solution: use binary search
  - (Or, even better, do resolutions offline after tracing)

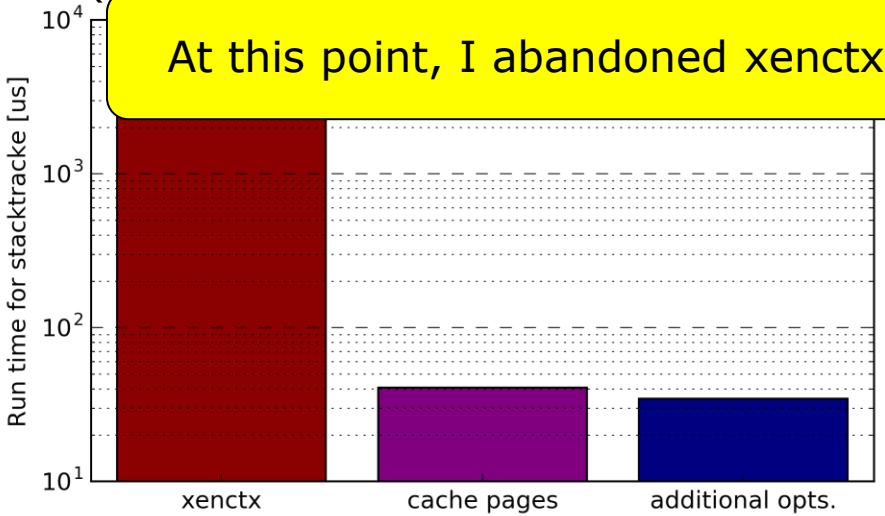


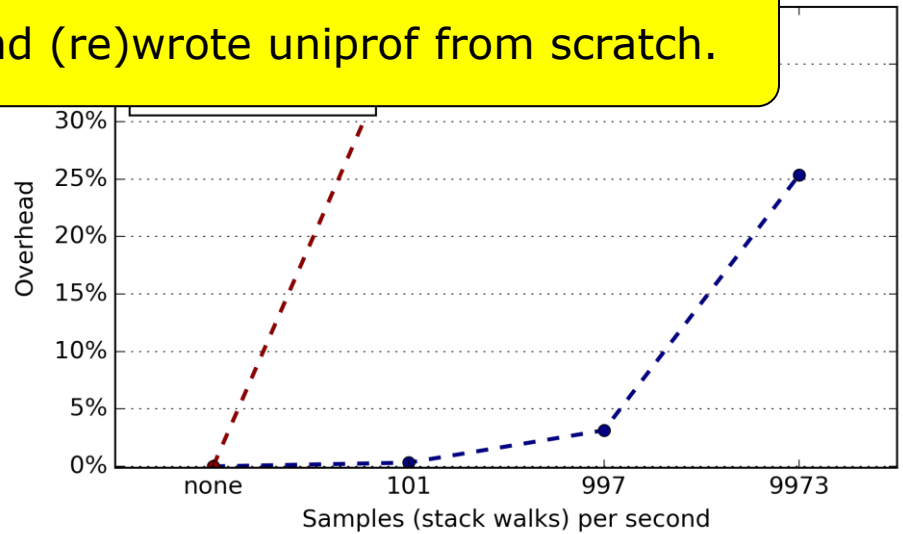


Try 1: improving xenctx performance

- xenctx translates and maps memory addresses every stack walk
  - Huge overhead
  - Solution: cache mapped memory and virtual→machine translations
- xenctx resolves symbols via linear search
  - Solution: use binary search
  - (Or, even better, do resolutions offline after tracing)

At this point, I abandoned xenctx and (re)wrote uniprof from scratch.



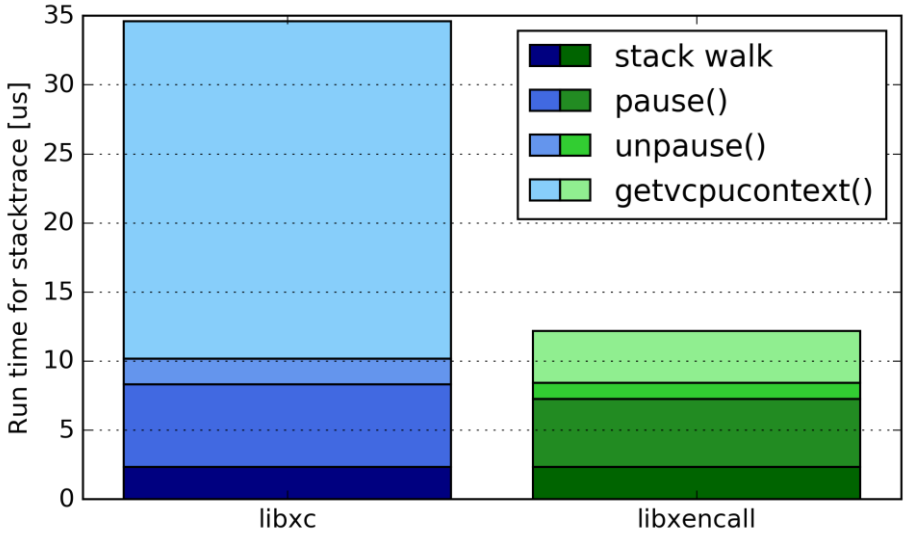


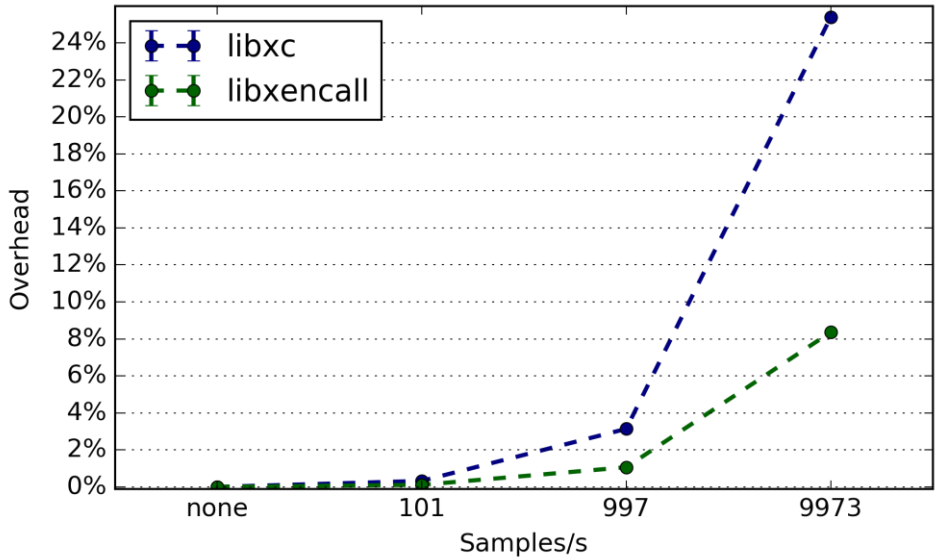
Try 2: uniprof

- 100-fold improvement is nice! But we can do better:
  - Xen 4.7 introduced low-level libraries (libxencall, libxenforeigmemory)
  - Another significant reduction by ~ factor of 3

Try 2: uniprof

- 100-fold improvement is nice! But we can do better:
  - Xen 4.7 introduced low-level libraries (libxencall, libxenforeigmemory)
  - Another significant reduction by ~ factor of 3
- End result: overhead of ~0.1% @101 samples/s



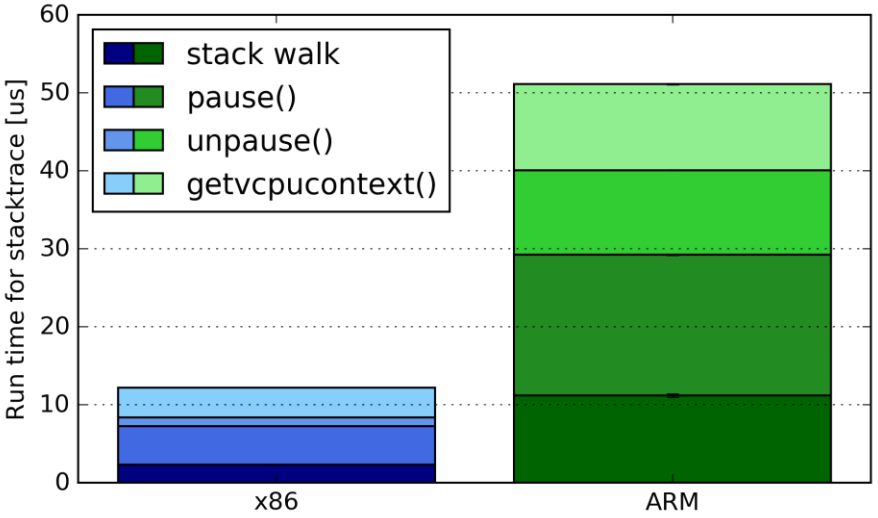


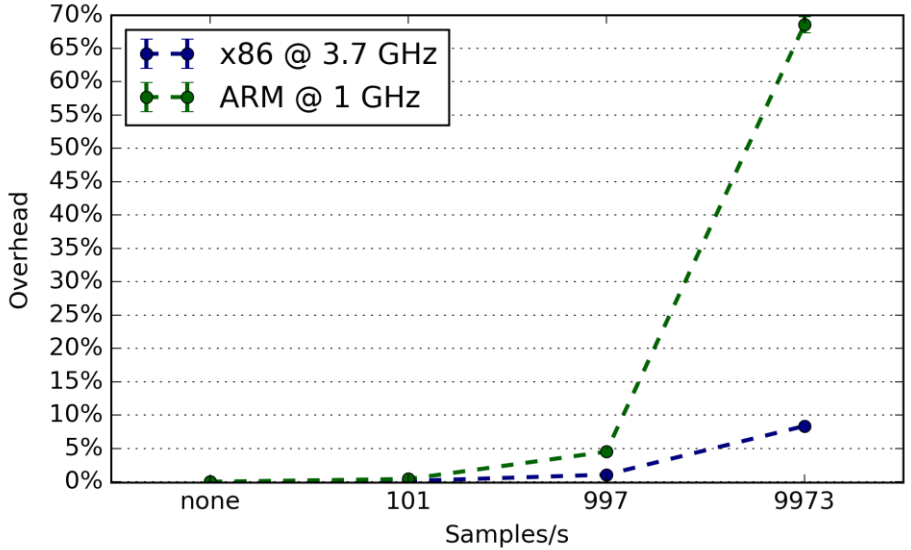
Performance on ARM

- uniprof supports ARM (xenctx doesn’t)
  - Main challenge: different page table design

Performance on ARM

- uniprof supports ARM (xenctx doesn't)
  - Main challenge: different page table design
- ARM: much slower, overhead higher
  - But the CPU is much slower, too (Intel Xeon @3.7GHz vs. Cortex A20 @1GHz)
  - So fewer samples/s needed for same effective resolution





No Frame Pointer? No Problem!

Stack walking relies on frame pointer

- Optimizations can reuse FP as general-purpose register (-fomit-frame-pointer)

No Frame Pointer? No Problem!

Stack walking relies on frame pointer

- Optimizations can reuse FP as general-purpose register (-fomit-frame-pointer)

But we can do without FPs

- Use stack unwinding information
  - It’s already included if you use C++ (for exception handling)
  - It doesn’t change performance
  - Only binary size
- DWARF standard

```
$ readelf -S <ELF>
There are 13 section headers, starting at offset 0x40d58:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
[...]
  [ 4] .eh_frame         PROGBITS         0000000000035860  00036860
       00000000000066f8  0000000000000000   A       0     0     8
  [ 5] .eh_frame_hdr     PROGBITS         000000000003bf58  0003cf58
       000000000000128c  0000000000000000   A       0     0     4
[...]
```

Unwinding without Frame Pointers

- How does it work?

Unwinding without Frame Pointers

How does it work?

Unwinding without Frame Pointers

How does it work?

Lookup table

- For every program address
  - The current frame size
  - Locations of registers to restore (GP and IP)

Unwinding without Frame Pointers

How does it work?

Lookup table

- For every program address
  - The current frame size
  - Locations of registers to restore (GP and IP)
- Important for exception handling
  - Exit functions immediately until handler is found

Unwinding without Frame Pointers

- How does it work?
- Lookup table
  - For every program address
    - The current frame size
    - Locations of registers to restore (GP and IP)
  - Important for exception handling
    - Exit functions immediately until handler is found
- Index to quickly find table entry

Unwinding without Frame Pointers

How does it work?

Lookup table

- For every program address
  - The current frame size
  - Locations of registers to restore (GP and IP)
- Important for exception handling
  - Exit functions immediately until handler is found

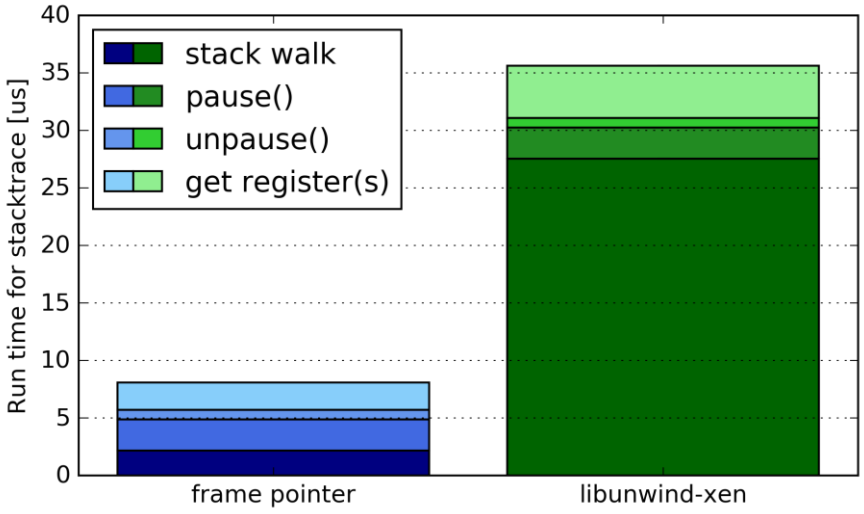
Index to quickly find table entry

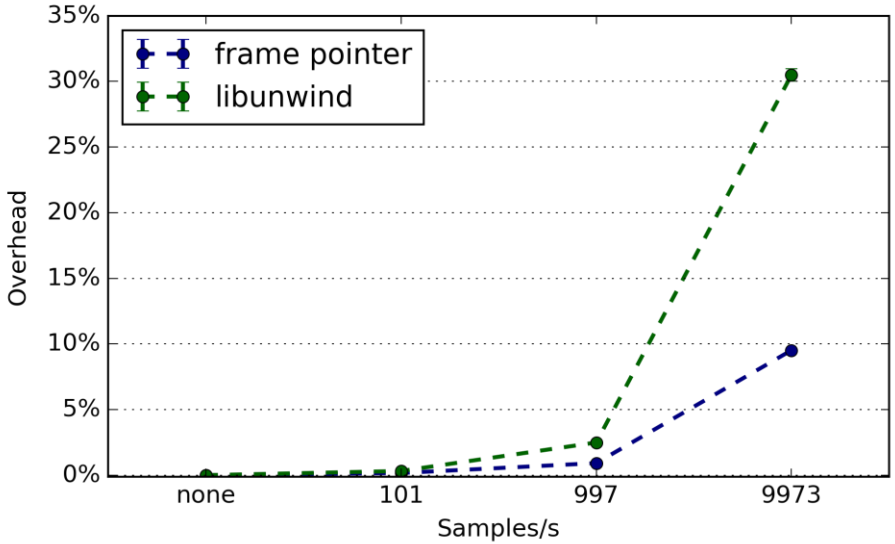
Several library implementations

- uniprof uses libunwind
- Actually, a libunwind patched for Xen guest introspection support
- Might be useful for other tools?

Performance: uniprof w/ libunwind

- Performance lower than with frame pointer
  - Reason: libunwind does more than we need (full register reconstruction etc.)
- Different library or own implementation promising
  - But "good enough" for many cases
  - And a good area for future work





Thank you!

Questions?

uniprof: https://github.com/cnplab/uniprof
libunwind-xen: https://github.com/cnplab/libunwind
FlameGraphs: https://github.com/brendangregg/flamegraph