ThreadSanitizer for OCaml

Olivier Nicole – Fabrice Buoro
(Tarides)

Goal of this talk

- What is ThreadSanitizer (TSan) and how is it useful?
- What is required to integrate the TSan runtime to OCaml programs?
- Hear your questions and suggestions about it

Finally, we can have data races too

A **data race** is a race condition defined by:

- Two accesses are made to the same memory location,
- At least one of them is a write, and
- No order is enforced between them.

Event ordering is formalized in terms of a partial order called *happens-before*. It is defined by the OCaml 5 memory model.

Data races are:

- Hard to detect (possibly silent)
- Hard to track down

ThreadSanitizer (TSan)

- **Runtime** data race detector (dynamic analysis, not static!)
- Initially developed for C++ by Google, now supported in
  - C, C++ with GCC and clang
  - Go
  - Swift
- Battle-tested, already found: [1]
  - 1200+ races in Google's codebase
  - ~100 in the Go stdlib
  - 100+ in Chromium
  - LLVM, GCC, OpenSSL, WebRTC, Firefox

- Requires to compile your program specially

1. Numbers August 2015

Demo

```
module Exercise (Q : Queueable) = struct
  let exercise queue =
    for i = 0 to 4 do
      Format.printf "Adding %d\n%!" i;
      Q.push i queue
    done

  let work () =
    let go = Atomic.make false in
    let q = Q.create () in
    let d = Domain.spawn (fun () -> Atomic.set go true; exercise q) in
    while not (Atomic.get go) do Domain.cpu_relax () done;
    exercise q;
    Domain.join d
end

module Seq = Exercise (Queue)
module Par = Exercise (struct
  include Lockfree.Michael_scott_queue
  let push i q = Fun.flip push i q
end)

let () =
  print_endline "With a non domain-safe queue";
  Seq.work ();
  print_endline "With a domain-safe queue";
  Par.work ()
```

How does it work?

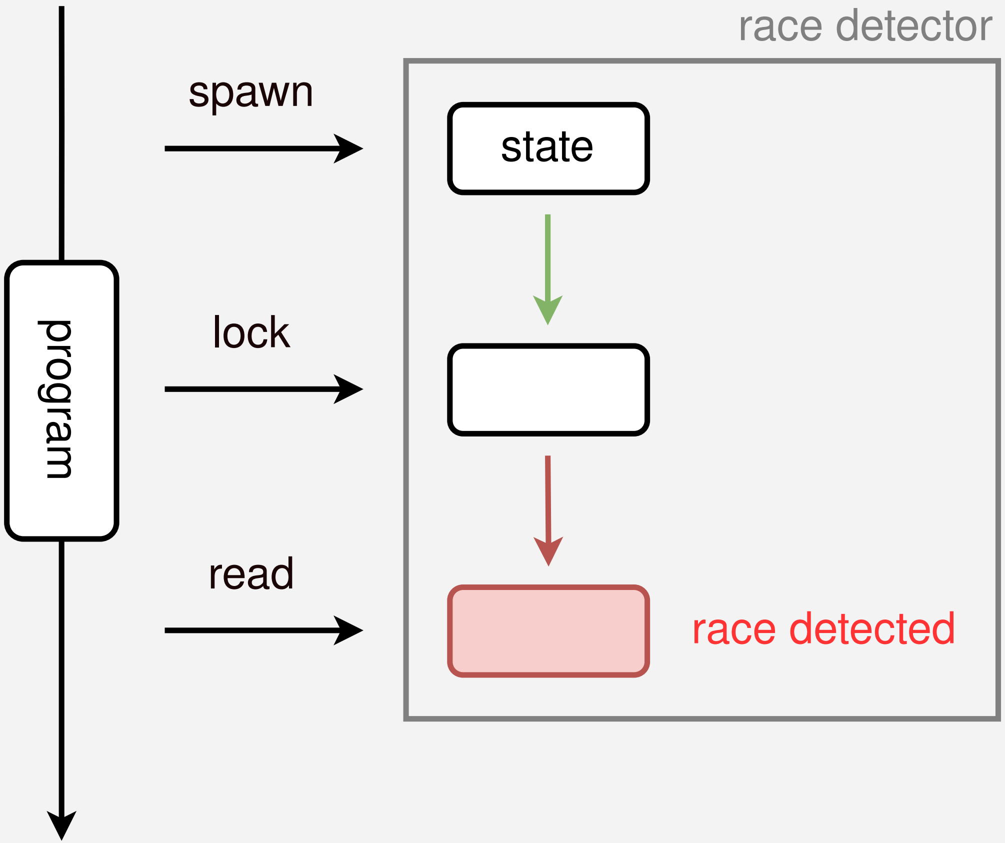
Two components

Program instrumentation

- Memory accesses
- Thread spawning and joining
- Mutex locks and unlocks, …

call →

Runtime library

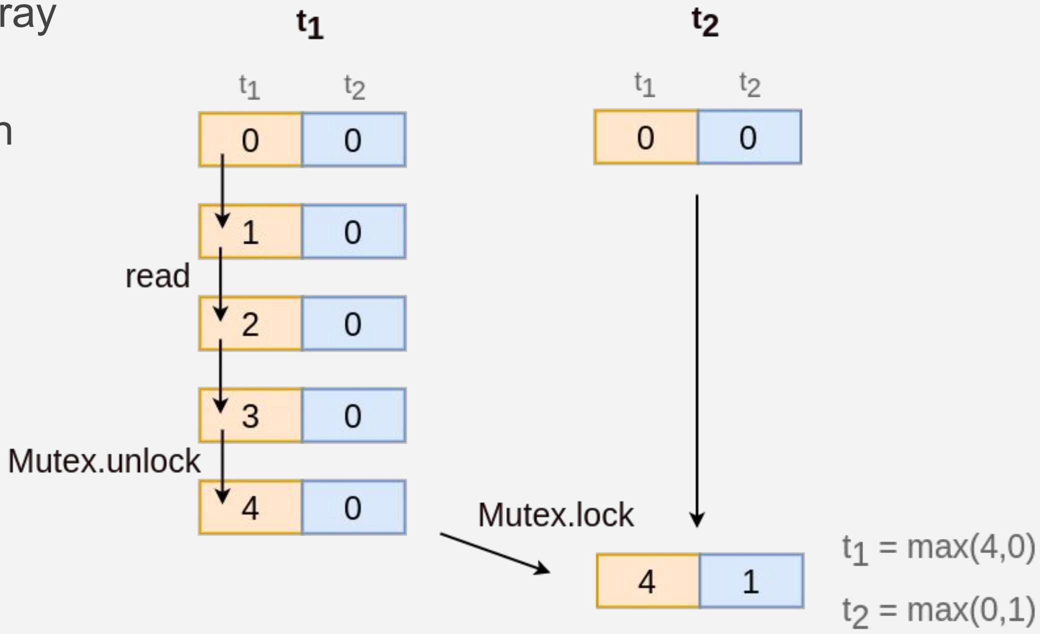


Race detector state machine

TSan's internal state

- Each thread holds a **vector clock** (array of *N* clocks, *N* = number of threads)
- Each thread increments its clock upon every **event** (memory access, mutex operation…)
- Some operations (e.g. mutex locks, atomic reads) synchronize clocks between threads

Comparing vector clocks allows to establish happens-before relations.



Shadow state

Stores information about memory accesses.

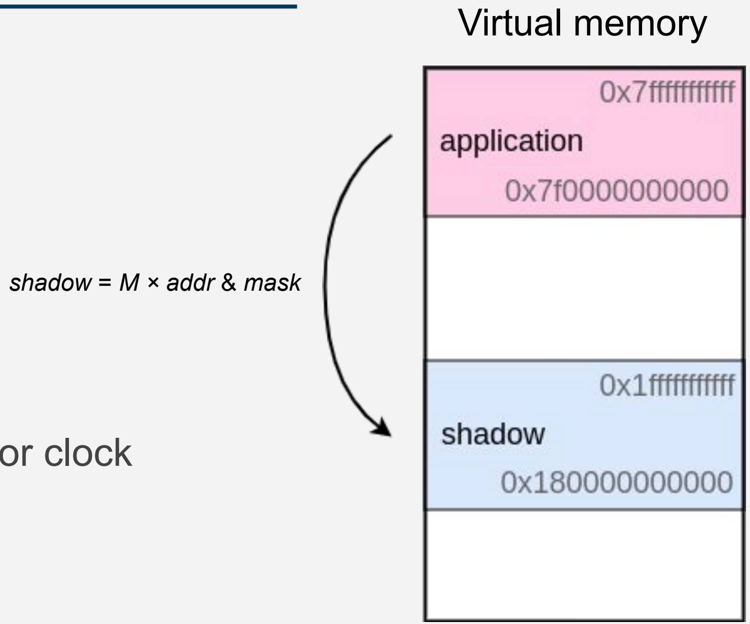
8-byte shadow word for an access:

TID: accessor thread ID

clock: scalar clock of accessor, optimized vector clock

pos: offset, size

w: is write



The shadow state stores M shadow words per application word ($M \in [2, 7]$, default $M = 4$)
If shadow words are filled, evict one at random

Race detection

Upon memory access, compare:

accessor's clock with each existing shadow word

- ❏ do the accesses overlap?
- ❏ is one of them a write?
- ❏ are the thread IDs different?
- ❏ are they unordered by happens-before?

Race detection

Upon memory access, compare:

accessor's clock with each existing shadow word

- ☑ do the accesses overlap?
- ☑ is one of them a write?
- ☑ are the thread IDs different?
- ☑ are they unordered by happens-before?

↓

RACE

Race detection

Upon memory access, compare:

accessor's clock with each existing shadow word

- ☑ do the accesses overlap?
- ☑ is one of them a write?
- ☑ are the thread IDs different?
- ☑ are they unordered by happens-before?

↓

RACE

Limitations:

- Runtime analysis: data races are only detected on visited code paths
- Finite number of memory accesses remembered (*M* per memory word)

So what do we need to support TSan?

Instrumentation of memory accesses

```
fun () ->
  r := 10;
  let x = !r in
  g x
```

Instrumentation of memory accesses

```
(function{simple_race.ml:6,24-59} camlSimple_race.fun_521
  (param/513: val)
 (store val r/503 21)



 (let x/514 (load_mut val r/503)



   (app{simple_race.ml:6,46-58} g/42 x/514 val))
```

```
fun () ->
  r := 10;
  let x = !r in
  g x
```

Instrumentation of memory accesses

```
(function{simple_race.ml:6,24-59} camlSimple_race.fun_521
  (param/513: val)
 (store val r/503 21)


 (let x/514 (load_mut val r/503)


   (app{simple_race.ml:6,46-58} g/42 x/514 val))
```

```
fun () ->
  r := 10;
  let x = !r in
  g x
```

Instrumentation of memory accesses

```
(function{simple_race.ml:6,24-59} camlSimple_race.fun_521
  (param/513: val)
 (store val r/503 21)


 (let x/514 (load_mut val r/503)



   (app{simple_race.ml:6,46-58} g/42 x/514 val))
```

→

```
(function{simple_race.ml:6,24-59} camlSimple_race.fun_521
  (param/513: val)
 (let (newval/531 21 loc/530 r/503)
   (extcall "__tsan_write8" loc/530 ->unit) 1
   (store val loc/530 newval/531))

 (let x/514
   (let loc/533 r/503
     (extcall "__tsan_read8" loc/533 ->unit) 1
     (load_mut val loc/533)))
   (app{simple_race.ml:7,47-59} g/42 x/514 val))
```

Instrumentation of memory accesses

```
(function{simple_race.ml:6,24-59} camlSimple_race.fun_521
  (param/513: val)
 (store val r/503 21)



 (let x/514 (load_mut val r/503)



   (app{simple_race.ml:6,46-58} g/42 x/514 val))
```

→

```
(function{simple_race.ml:6,24-59} camlSimple_race.fun_521
  (param/513: val)
 (let (newval/531 21 loc/530 r/503)
   (extcall "__tsan_write8" loc/530 ->unit) 1
   (store val loc/530 newval/531))

 (let x/514
   (let loc/533 r/503
     (extcall "__tsan_read8" loc/533 ->unit) 1
     (load_mut val loc/533)))
   (app{simple_race.ml:7,47-59} g/42 x/514 val))
```

- In OCaml, writes are done through `caml_modify` (except for immediates), so it needs to be instrumented too
- In general, runtime C functions that do significant things (memory accesses, thread operations…) need to be instrumented
  - We use the built-in TSan support in gcc/clang to instrument them

Function entries and exits

- Recall: TSan gives the backtrace of **both** conflicting accesses

```
==================
WARNING: ThreadSanitizer: data race (pid=4170290)
  Read of size 8 at 0x7f072bbfe498 by thread T4 (mutexes: write M0):
    #0 camlSimpleRace__fun_524 /tmp/simpleRace.ml:7 (simpleRace.exe+0x43dc9d)
    #1 camlStdlib__Domain__body_696 /home/olivier/.opam/5.0.0+tsan/.opam-switch/build/ocaml-variants.5.
    #2 caml_start_program ??:? (simpleRace.exe+0x4f51c3)
    #3 caml_callback_exn /home/olivier/.opam/5.0.0+tsan/.opam-switch/build/ocaml-variants.5.0.0+tsan/ru
    #4 caml_callback /home/olivier/.opam/5.0.0+tsan/.opam-switch/build/ocaml-variants.5.0.0+tsan/runtim
    #5 domain_thread_func /home/olivier/.opam/5.0.0+tsan/.opam-switch/build/ocaml-variants.5.0.0+tsan/r

  Previous write of size 8 at 0x7f072bbfe498 by thread T1 (mutexes: write M1):
    #0 camlSimpleRace__fun_520 /tmp/simpleRace.ml:6 (simpleRace.exe+0x43dc45)
    #1 camlStdlib__Domain__body_696 /home/olivier/.opam/5.0.0+tsan/.opam-switch/build/ocaml-variants.5.
    #2 caml_start_program ??:? (simpleRace.exe+0x4f51c3)
    #3 caml_callback_exn /home/olivier/.opam/5.0.0+tsan/.opam-switch/build/ocaml-variants.5.0.0+tsan/ru
    #4 caml_callback /home/olivier/.opam/5.0.0+tsan/.opam-switch/build/ocaml-variants.5.0.0+tsan/runtim
    #5 domain_thread_func /home/olivier/.opam/5.0.0+tsan/.opam-switch/build/ocaml-variants.5.0.0+tsan/r

  Mutex M0 (0x000000567ad8) created at:
    #0 pthread_mutex_init /home/olivier/other_projects/llvm-project/compiler-rt/lib/tsan/rtl/tsan_inter
    [...]

SUMMARY: ThreadSanitizer: data race /tmp/simpleRace.ml:7 in camlSimpleRace__fun_524
==================
ThreadSanitizer: reported 1 warnings
```

Function entries and exits

```
(function{simple_race.ml:6,24-59} camlSimple_race.fun_521
  (param/513: val)

 (let (newval/531 21 loc/530 r/503)
   (extcall "__tsan_write8" loc/530 ->unit) 1
   (store val loc/530 newval/531))
 (let x/514
   (let loc/533 r/503
     (extcall "__tsan_read8" loc/533 ->unit) 1
     (load_mut val loc/533)))


   (app{simple_race.ml:7,47-59} g/42 x/514 val))
```

```
(function{simple_race.ml:6,24-59} camlSimple_race.fun_521
  (param/513: val)
 (extcall "__tsan_func_entry" return_addr ->unit) 1
 (let (newval/531 21 loc/530 r/503)
   (extcall "__tsan_write8" loc/530 ->unit) 1
   (store val loc/530 newval/531))
 (let x/514
   (let loc/533 r/503
     (extcall "__tsan_read8" loc/533 ->unit) 1
     (load_mut val loc/533)))
 (let arg/532 x/514
   (extcall "__tsan_func_exit" ->unit) 1
   (app{simple_race.ml:6,46-58} g/42 arg/532 val)))
```

- To be able to show backtraces of past program points, TSan requires us to instrument function entries and exits
- Tail calls must be handled with care

Technical point #1.1 Exceptions

- In C, it is easy to instrument function entry and exits
- C++ has to take care of exceptions
- In OCaml also:
  - Any function can be exited due to an exception
  - Unlike in C++, exceptions do not unwind the stack

- TSan's linear view of the call stack does not hold.

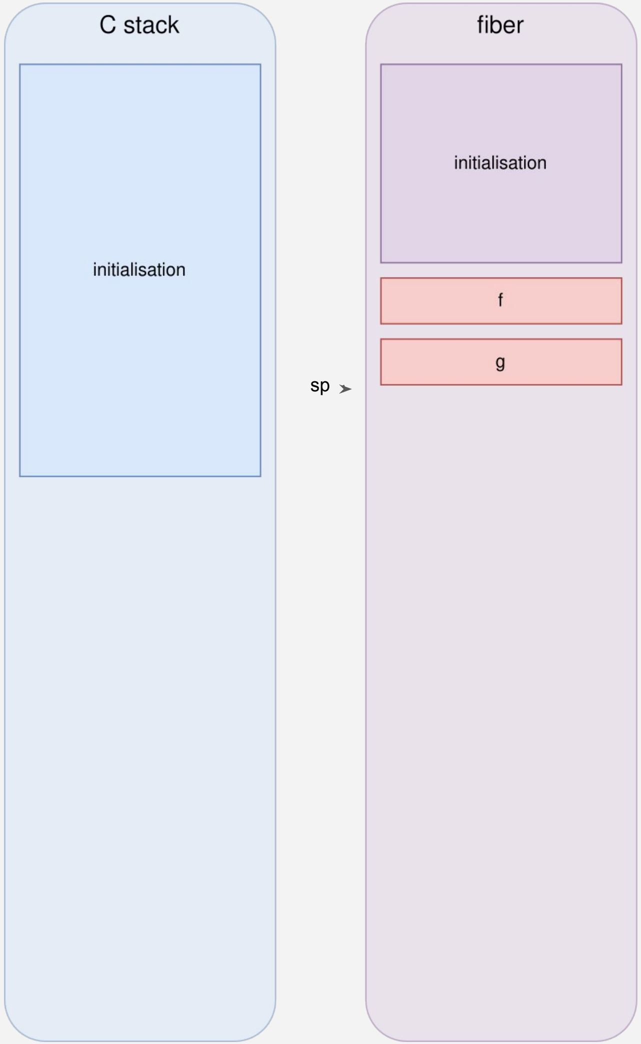
Technical point #1.1 Exceptions

```
let i () =  raise MyExn

let h () =  i ()

let g () = print_and_call_ocaml_h ()

let f () =
  try g () with
  | MyExn -> race ()

let () =
  let d = Domain.spawn (fun () -> race ()) in
  f ();
  Domain.join d
```

```
value print_and_call_ocaml_h(value unit)
{
 printf("Hello from C\n");
 caml_callback(*caml_named_value("h"), Val_unit);
 return Val_unit;
}
```

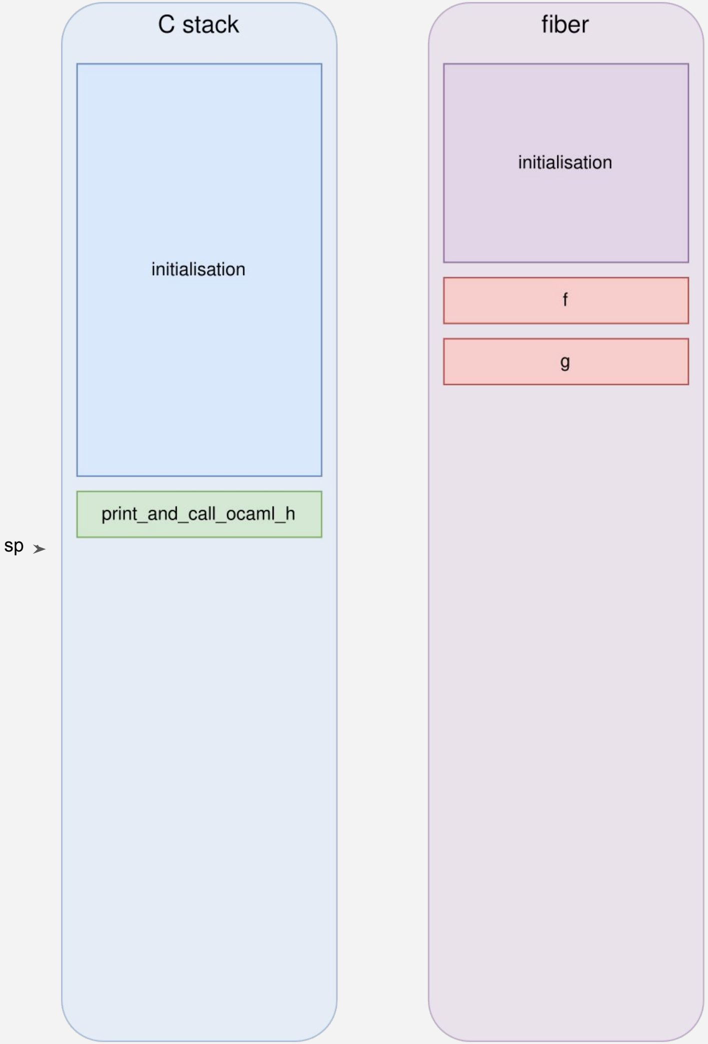
- **Cmm** instrumentation emits call to `tsan_func_entry` when entering a function
- TSan backtrace:
  - f
  - g

```
let i () = raise MyExn

let h () =  i ()

let g () = print_and_call_ocaml_h ()

let f () =
  try g () with
  | MyExn -> race ()

let () =
  let d = Domain.spawn (fun () -> race ()) in
  f ();
  Domain.join d
```



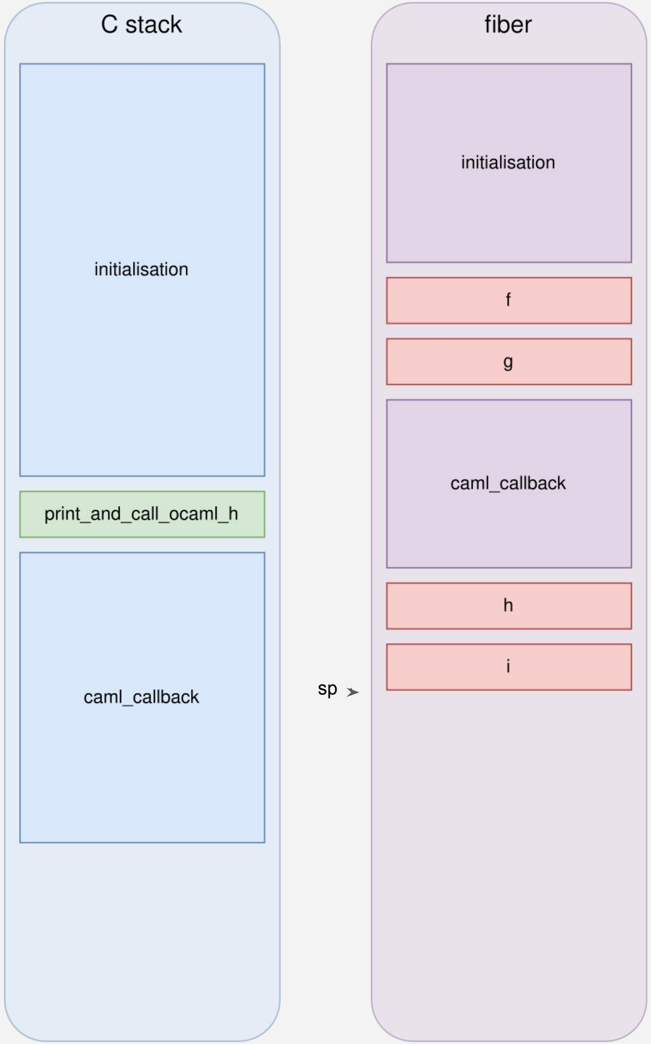
- Switching back to C stack for the C function call
- C code is instrumented by the C compiler which also emits call to `tsan_func_entry` on function entry
- TSan backtrace:
  - `f`
  - `g`
  - `print_and_call_ocaml_h`

```
let i () = raise MyExn

let h () =  i ()

let g () = print_and_call_ocaml_h ()

let f () =
  try g () with
  | MyExn -> race ()

let () =
  let d = Domain.spawn (fun () -> race ()) in
  f ();
  Domain.join d
```



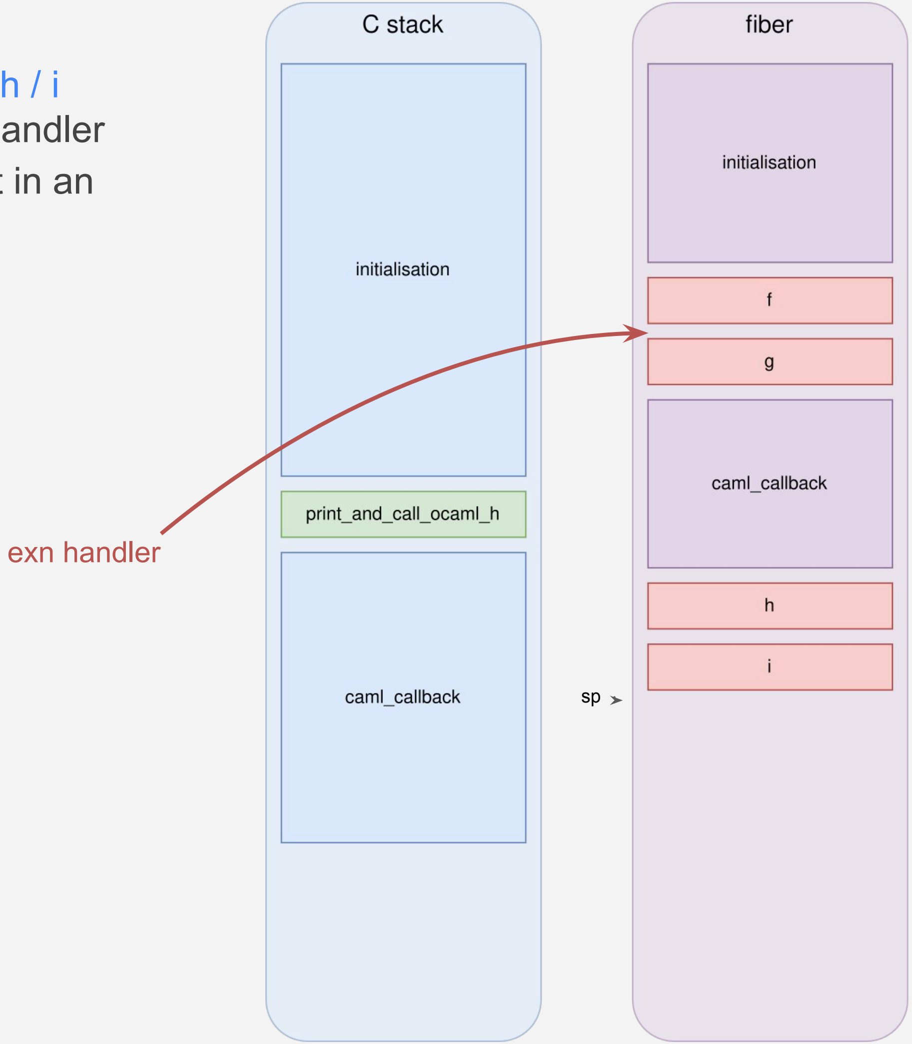
- Switching back to OCaml stack for the callback
- TSan backtrace:
    - `f`
    - `g`
    - `print_and_call_ocaml_h`
    - `h`
    - `i`

```
let i () = raise MyExn

let h () =  i ()

let g () = print_and_call_ocaml_h ()

let f () =
  try g () with
  | MyExn -> race ()

let () =
  let d = Domain.spawn (fun () -> race ()) in
  f ();
  Domain.join d
```



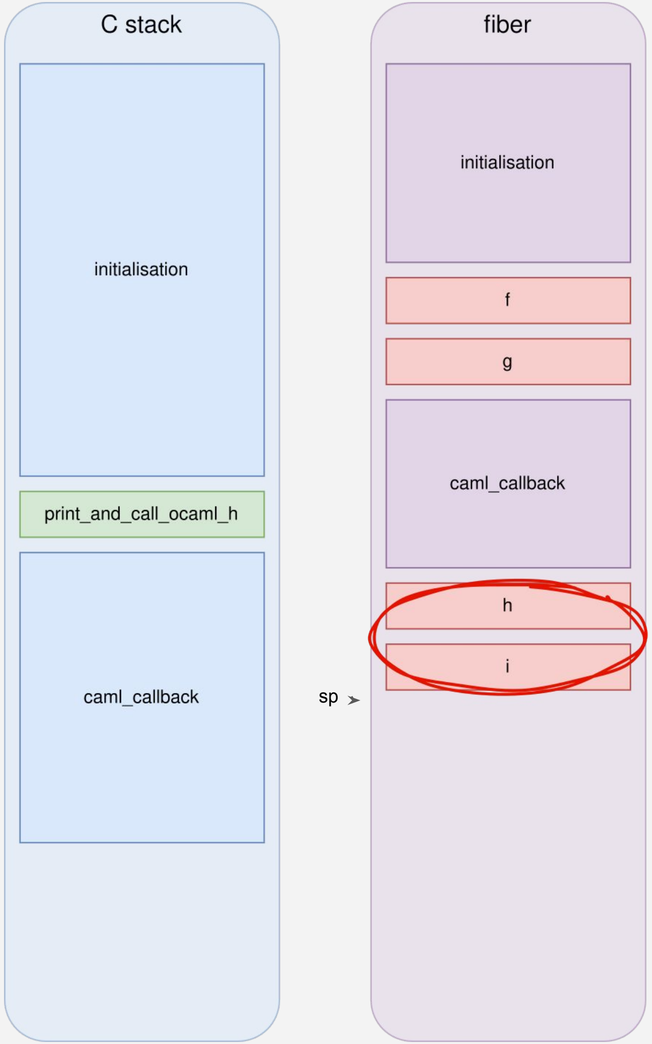
- For TSan, we are still in f / g / print_and_call_h / h / i
    - Calling the race function of the exception handler without any other prior actions would result in an incorrect backtrace

```
let i () = raise MyExn

let h () =  i ()

let g () = print_and_call_ocaml_h ()

let f () =
  try g () with
  | MyExn -> race ()

let () =
  let d = Domain.spawn (fun () -> race ()) in
  f ();
  Domain.join d
```



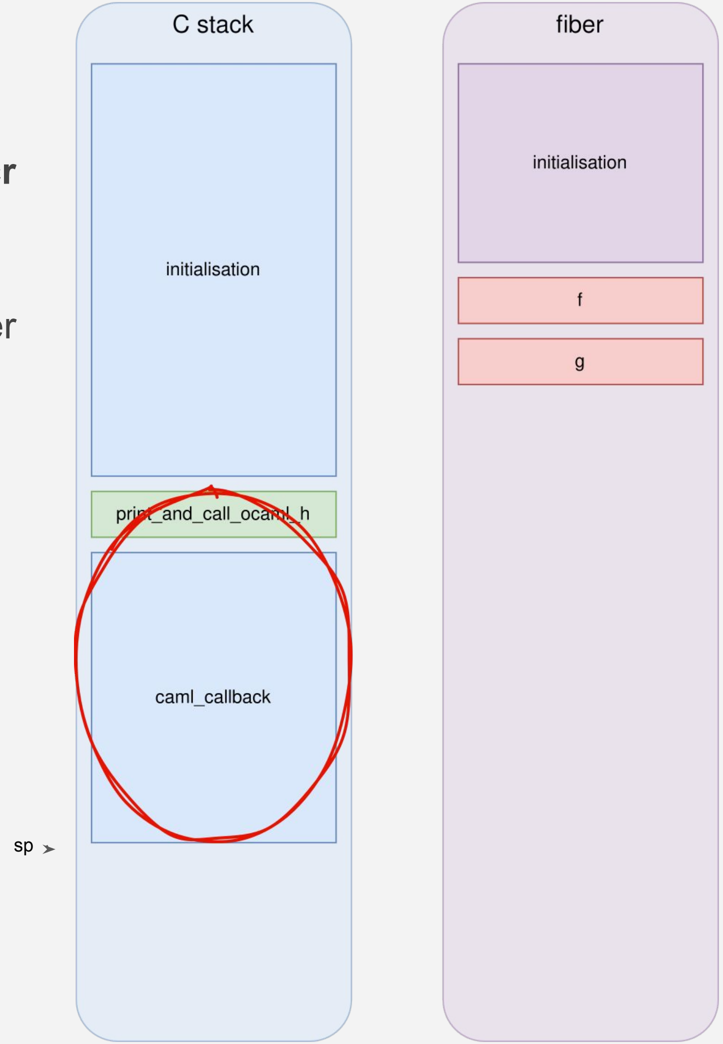
- For TSan, we are still in f / g / print_and_call_h / h / i
  - Calling the race function of the exception handler without any other prior actions would result in an incorrect backtrace
- While raising the exception, in caml_raise_exn
  - Use **frame_descr** to scan the stack up to the next exception handler
  - Emit tsan_func_exit for every stack frame

```
let i () = raise MyExn

let h () =  i ()

let g () = print_and_call_ocaml_h ()

let f () =
  try g () with
  | MyExn -> race ()

let () =
  let d = Domain.spawn (fun () -> race ()) in
  f ();
  Domain.join d
```



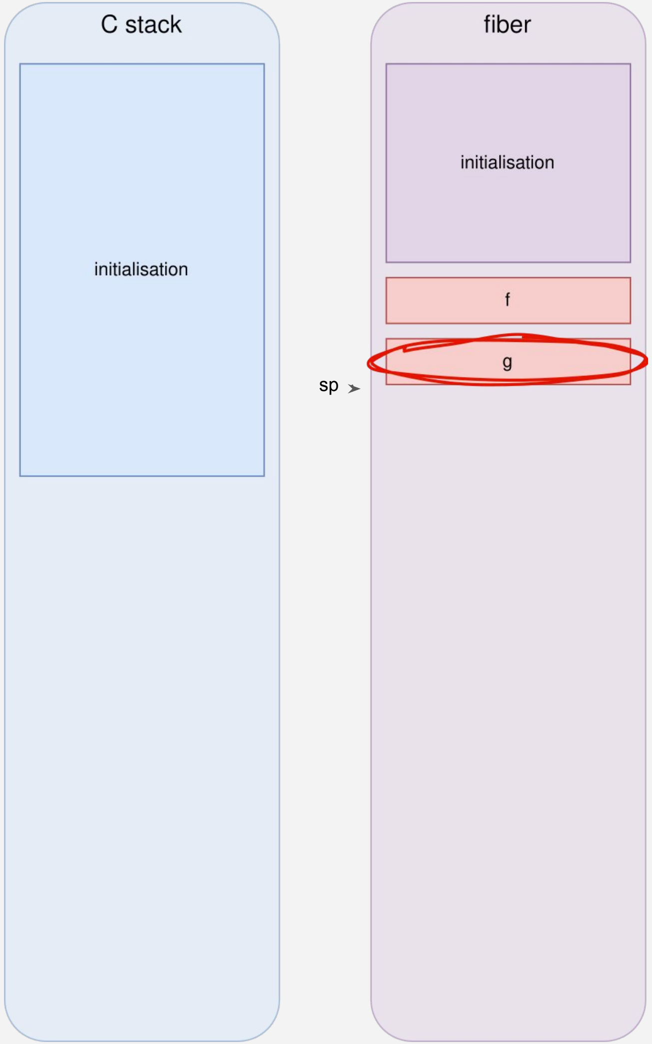
- For TSan, we are still in `f / g / print_and_call_h`
- The exception propagates through the C stack, **frame_descr** can't help here
- In `caml_raise`
  - Use **libunwind** to scan the stack up to the next handler
  - Emit `tsan_func_exit` for every C stack frame

```
let i () = raise MyExn

let h () =  i ()

let g () = print_and_call_ocaml_h ()

let f () =
  try g () with
  | MyExn -> race ()

let () =
  let d = Domain.spawn (fun () -> race ()) in
  f ();
  Domain.join d
```



- Again in the OCaml stack
- The process repeat: back to using **frame_descr** in `caml_raise_exn` to emit `tsan_func_exit` until the exception handler (in function `f`)

```
let i () = raise MyExn

let h () =  i ()

let g () = print_and_call_ocaml_h ()

let f () =
  try g () with
  | MyExn -> race ()

let () =
  let d = Domain.spawn (fun () -> race ()) in
  f ();
  Domain.join d
```



Technical point #1.2 Effect handlers

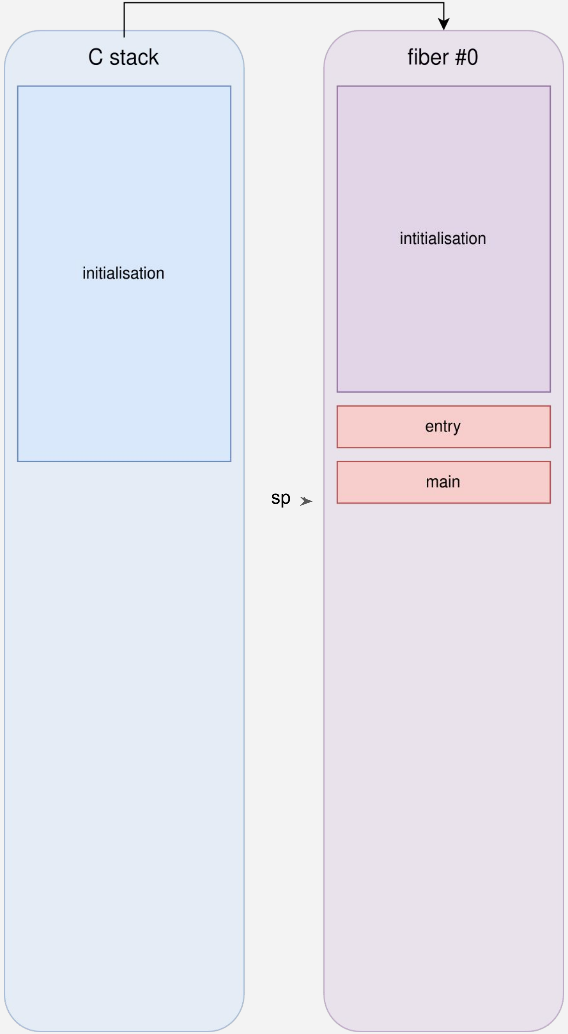
- Effect handlers are like exceptions, except you can come back

```
type _ Effect.t += E : string Effect.t

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  match_with comp () {
    retc = Fun.id;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | E -> Some (fun (k : (a, unit) continuation) ->
          print_string "1 "; continue k "2 "; print_string "4 ")
      | _ -> None);
    exnc = (fun e -> raise e); }
```

https://kcsrk.info/slides/retro_effects_simcorp.pdf

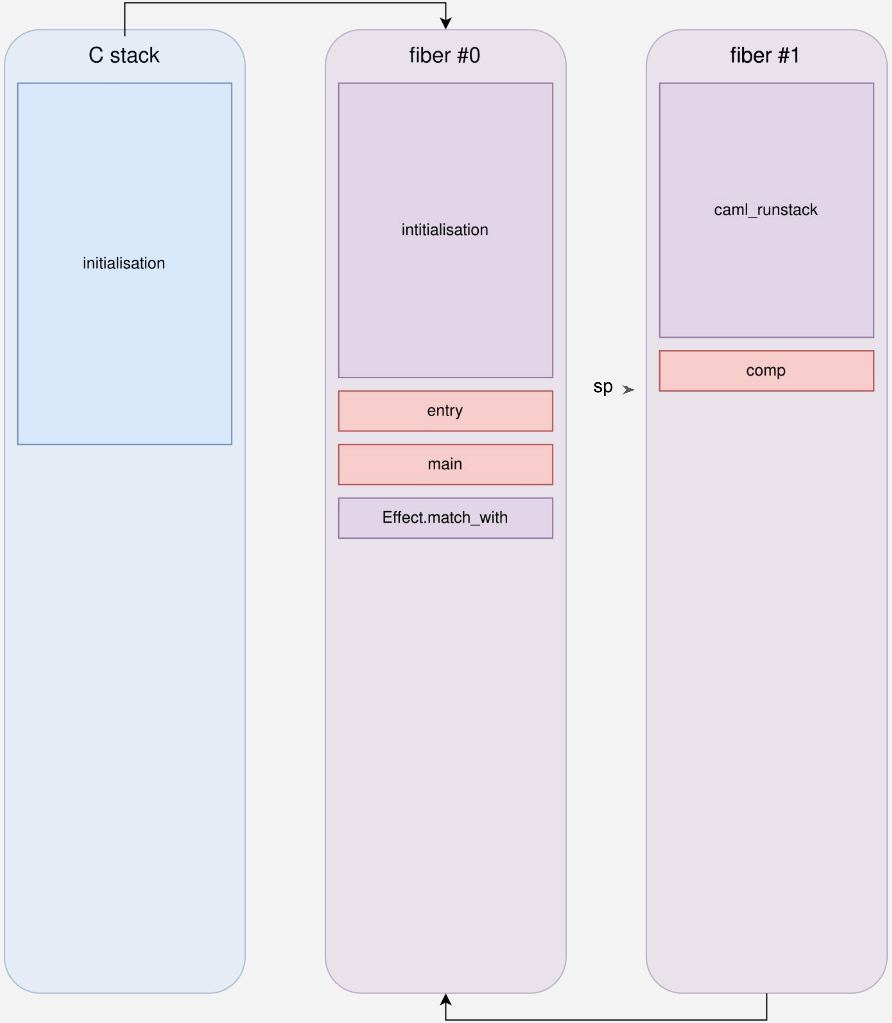
- OCaml startup spawns the initial fiber

```
let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  match_with comp () {
    retc = Fun.id;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | E -> Some (fun (k : (a, unit) continuation) ->
          print_string "1 "; continue k "2 "; print_string "4 ")
      | _ -> None);
    exnc = (fun e -> raise e); }
```



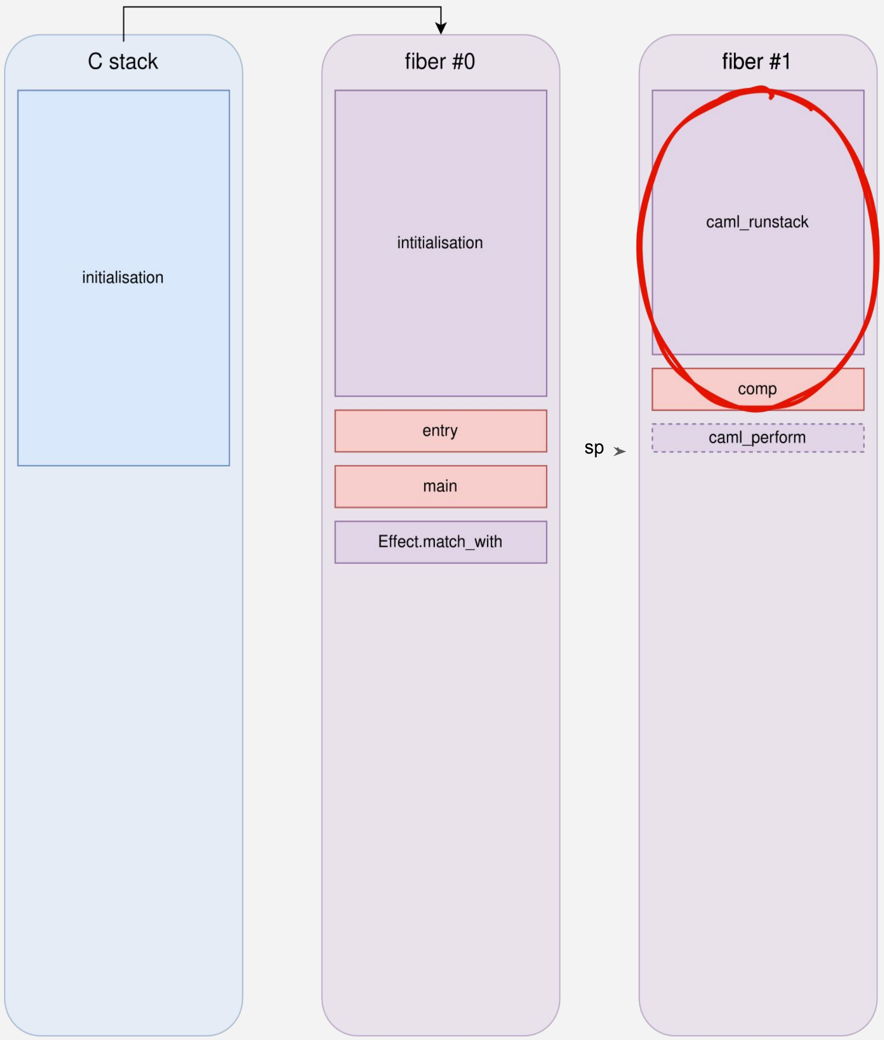
- `main` calls `Effect.match_with`
  - Allocates a new fiber
  - Switches to the stack into fiber #1
  - Executes the computation (through `caml_runstack`)

```
let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  match_with comp () {
    retc = Fun.id;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | E -> Some (fun (k : (a, unit) continuation) ->
          print_string "1 "; continue k "2 "; print_string "4 ")
      | _ -> None);
    exnc = (fun e -> raise e); }
```



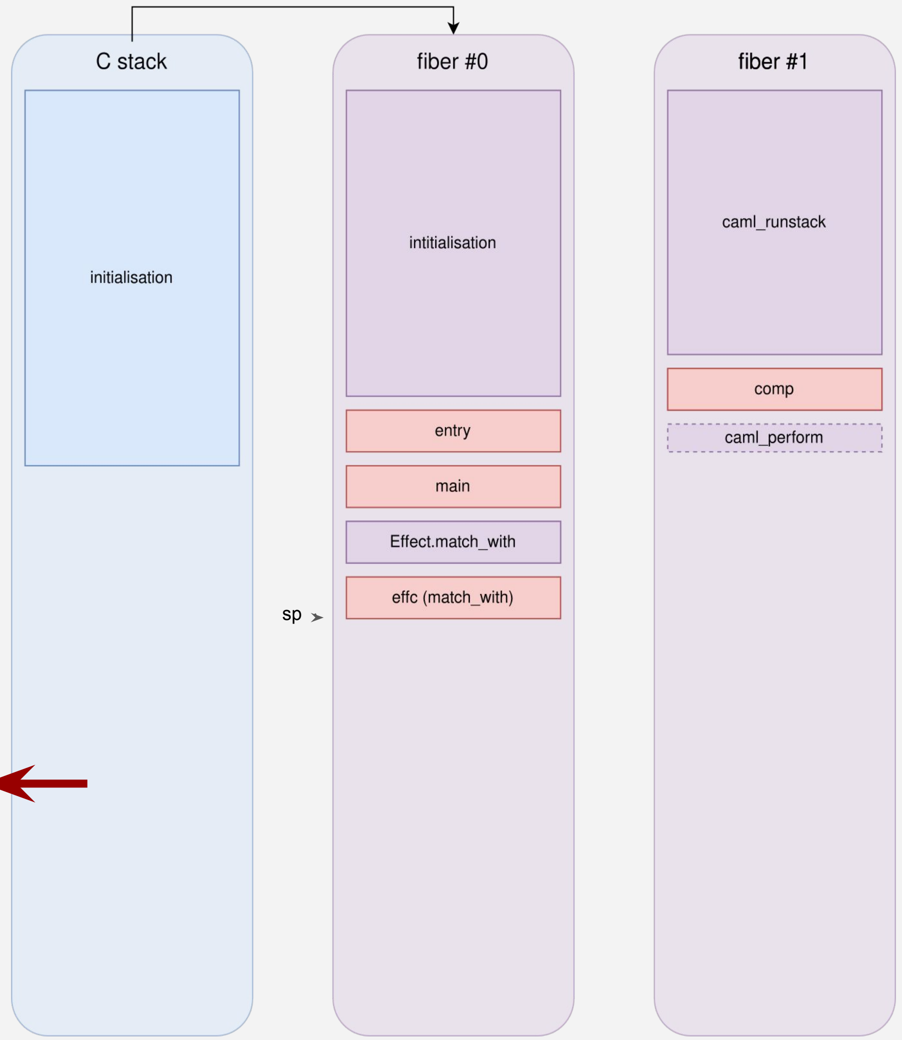
- Perform the E effect
- caml_perform
    - In order to resume execution into the effect handler of fiber #0
    - Use **frame_descr** to emit calls to tsan_func_exit

```
let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  match_with comp () {
    retc = Fun.id;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | E -> Some (fun (k : (a, unit) continuation) ->
          print_string "1 "; continue k "2 "; print_string "4 ")
      | _ -> None);
    exnc = (fun e -> raise e); }
```



- Into the effect handler `effc` from fiber #0

```
let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  match_with comp () {
    retc = Fun.id;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | E -> Some (fun (k : (a, unit) continuation) ->
          print_string "1 "; continue k "2 "; print_string "4 ")
      | _ -> None);
    exnc = (fun e -> raise e); }
```

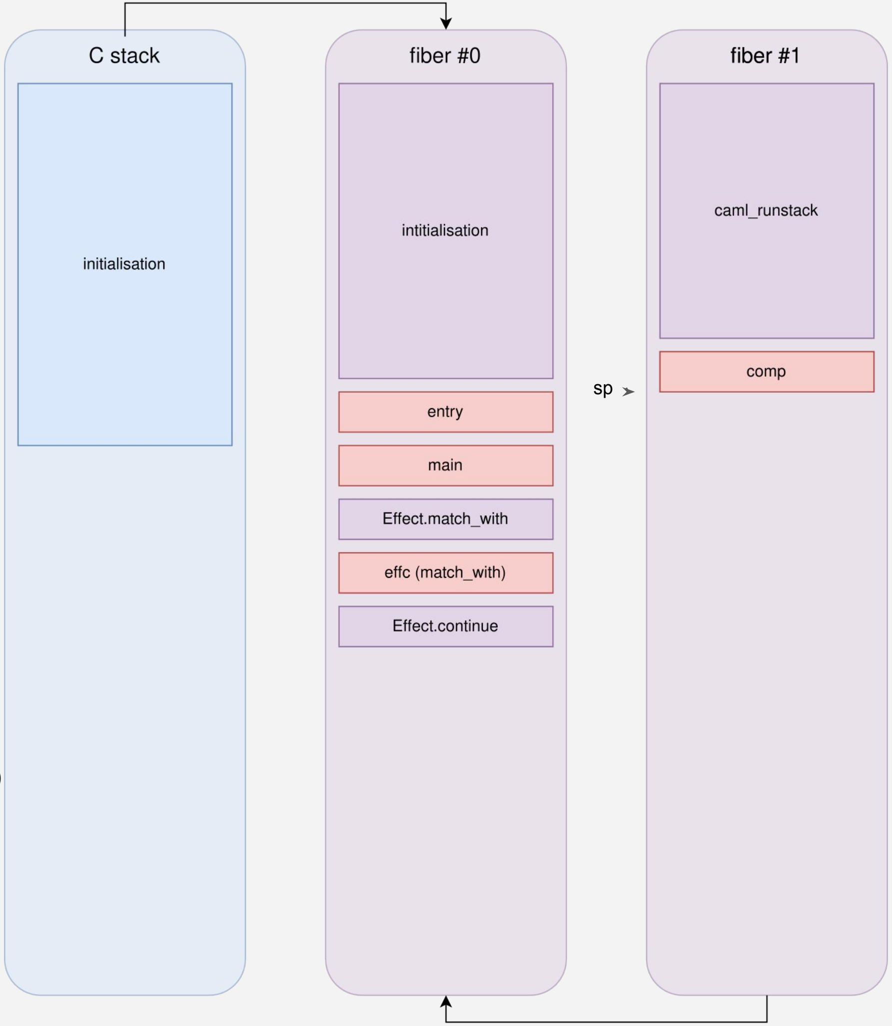


- Calls continue to resume execution in the computation
- `caml_resume`
    - In order to resume execution in the fiber #1 stack
    - Use `frame_descr` to emit calls to `tsan_func_entry`

```
let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  match_with comp () {
    retc = Fun.id;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | E -> Some (fun (k : (a, unit) continuation) ->
          print_string "1 "; continue k "2 "; print_string "4 ")
      | _ -> None);
    exnc = (fun e -> raise e); }
```

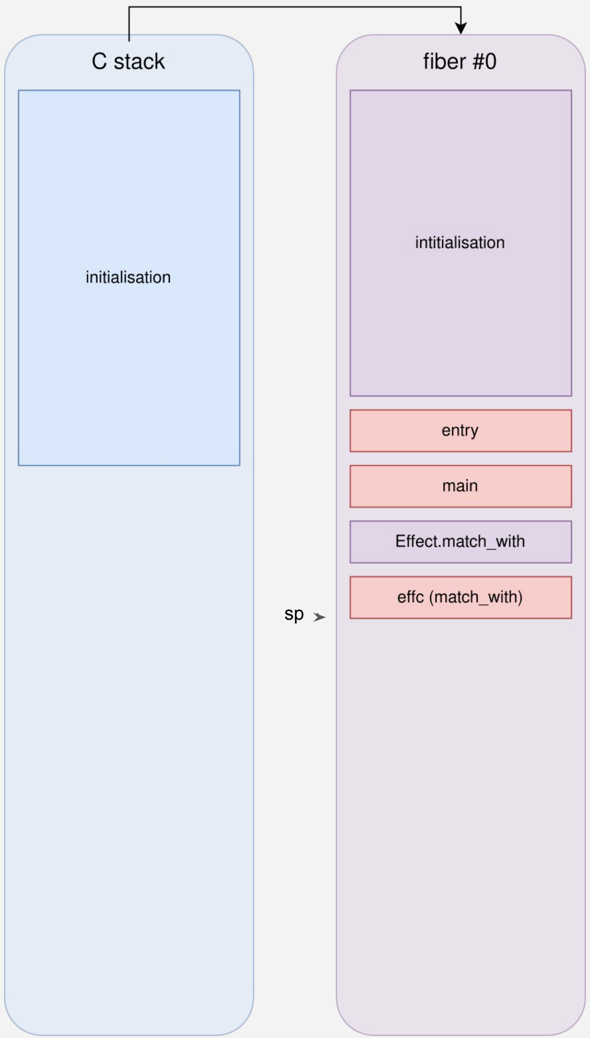
- The computation completes
- `caml_runstack`
  - Free the fiber
  - Resume execution in the initial fiber
  - Call the value handler

```
let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  match_with comp () {
    retc = Fun.id;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | E -> Some (fun (k : (a, unit) continuation) ->
          print_string "1 "; continue k "2 "; print_string "4 ")
      | _ -> None);
    exnc = (fun e -> raise e); }
```



- Completes the effect handler and so the `match_with`

```
let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  match_with comp () {
    retc = Fun.id;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | E -> Some (fun (k : (a, unit) continuation) ->
          print_string "1 "; continue k "2 "; print_string "4 ")
      | _ -> None);
    exnc = (fun e -> raise e); }
```



Technical point #2: Memory model

- TSan understands the **C11 memory model**
- The OCaml 5 memory model is quite different

We map OCaml memory accesses to C11 accesses. The mapping must be such that:

- Racy programs (in the OCaml sense) must be mapped to racy programs (in the C11 sense) so that OCaml data races are detected
- Race-free programs (in the OCaml sense) must be mapped to race-free programs (in the C11 sense) as we don’t want false positives

⇨ What we “show” to TSan is not necessarily the real memory operations.

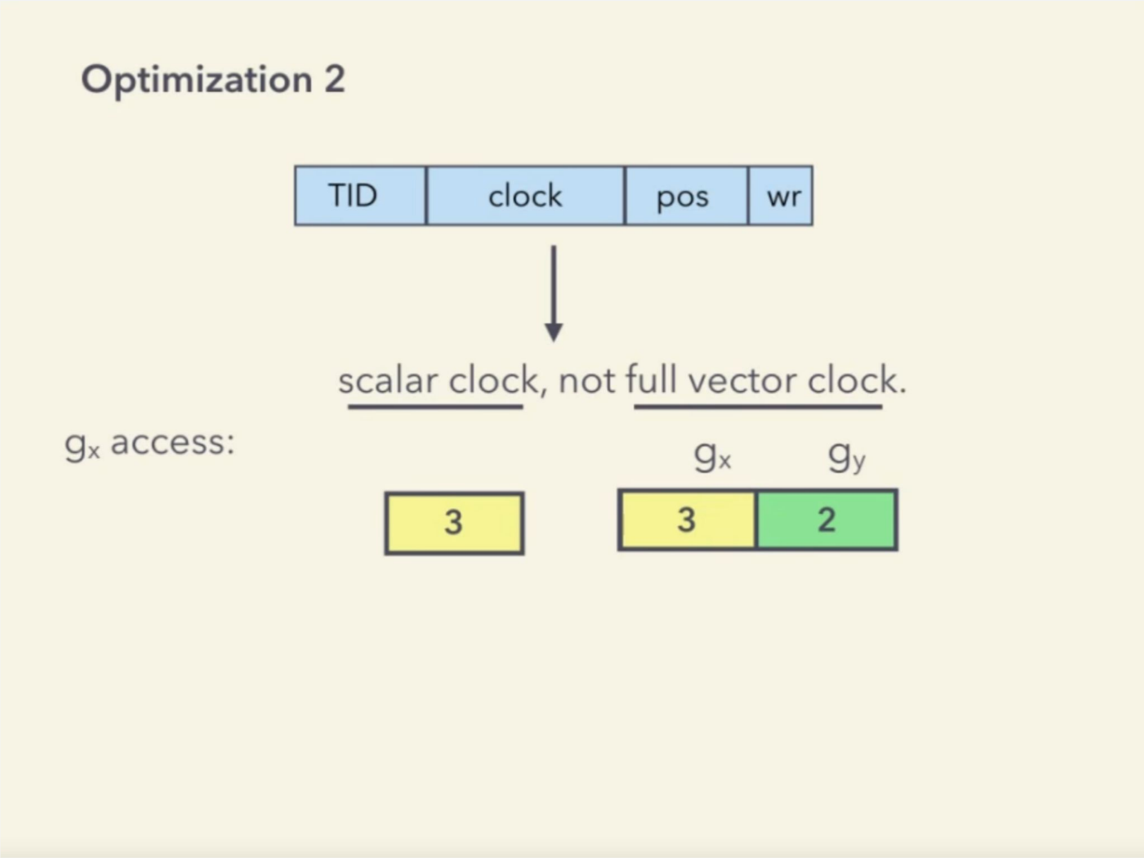
Operation	Location in the codebase	Implementation	TSan view
Atomic load	`caml_atomic_load`	fence(acquire) atomic_load(seq_cst)	atomic_load(seq_cst)
Atomic store	`caml_atomic_exchange`	fence(acquire) atomic_exchange(seq_cst) fence(release)	atomic_exchange(seq_cst)
Non-atomic load	assembly	atomic_load(relaxed)	plain load
Non-atomic store (initializing)	assembly or `caml_initialize`	plain store	-
Non-atomic store (assignment, integer)	assembly or `caml_modify`	fence(acquire) atomic_store(release)	plain store
Non-atomic store (assignment, pointer)	assembly or `caml_modify`	fence(acquire) atomic_store(release)	plain store
Non-atomic store (non-word-sized field)	assembly	plain store	plain store

Operation	Location in the codebase	Implementation	TSan view
Atomic load	`caml_atomic_load`	fence(acquire) atomic_load(seq_cst)	atomic_load(seq_cst)
Atomic store	`caml_atomic_exchange`	fence(acquire) atomic_exchange(seq_cst) fence(release)	atomic_exchange(seq_cst)
Non-atomic load	assembly	atomic_load(relaxed)	plain load
Non-atomic store (initializing)	assembly or `caml_initialize`	plain store	-
Non-atomic store (assignment, integer)	assembly or `caml_modify`	fence(acquire) atomic_store(release)	plain store
Non-atomic store (assignment, pointer)	assembly or `caml_modify`	fence(acquire) atomic_store(release)	plain store
Non-atomic store (non-word-sized field)	assembly	plain store	plain store

Current status

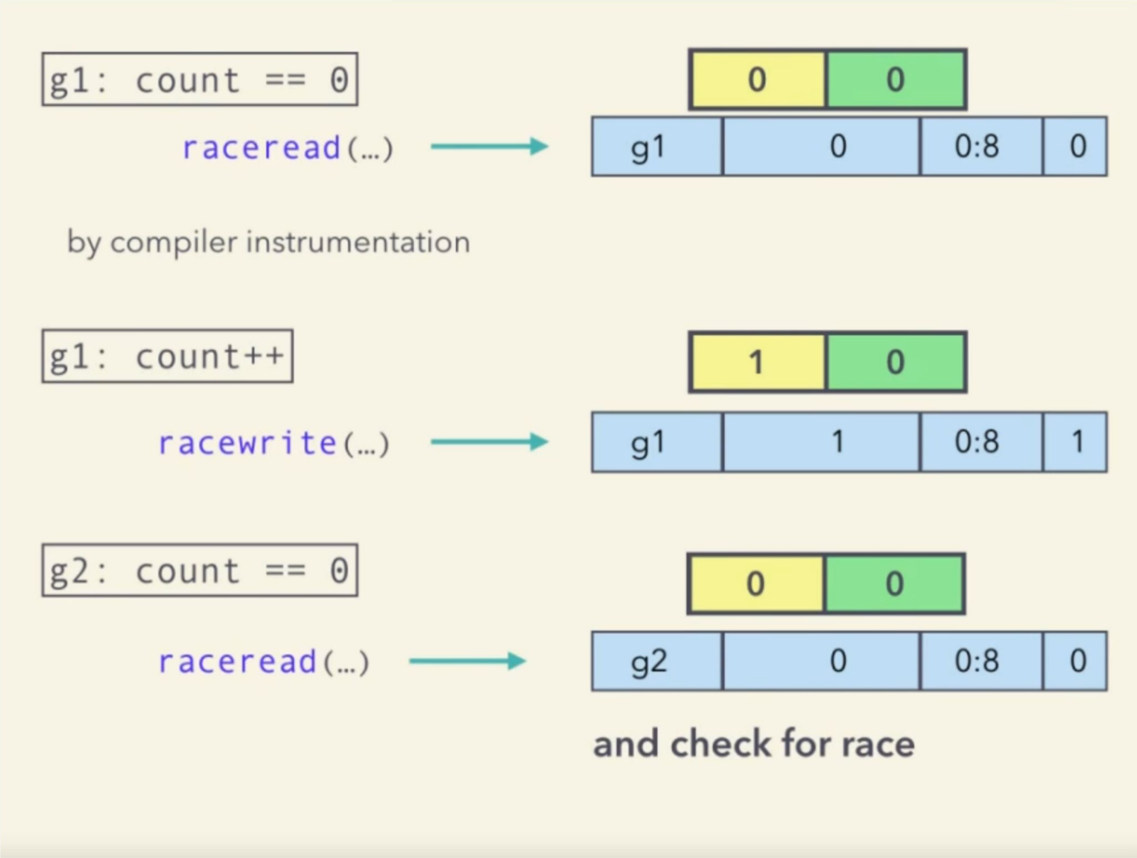
- The instrumentation has a performance cost: about 7-13x slowdown
  - compared to 5-15x for C/C++
- Memory consumption is increased by 2-7x (compared to 5-10x for C/C++)
- No cost if TSan is not enabled on your opam switch
- An earlier version based on OCaml 5.0 is already available on opam:
  `opam switch create 5.0.0+tsan`
- We have already used the mode to find races in
  - Lockfree: ocaml-multicore/lockfree#40, ocaml-multicore/lockfree#39
  - Domainslib: ocaml-multicore/domainslib#72, ocaml-multicore/domainslib#103
  - The OCaml runtime: ocaml/ocaml#11040
- A feature complete PR is ready: ocaml/ocaml#12114
  - ~1,700 lines of diff + 1,000 lines of test suite
  - No full review yet

Thank You

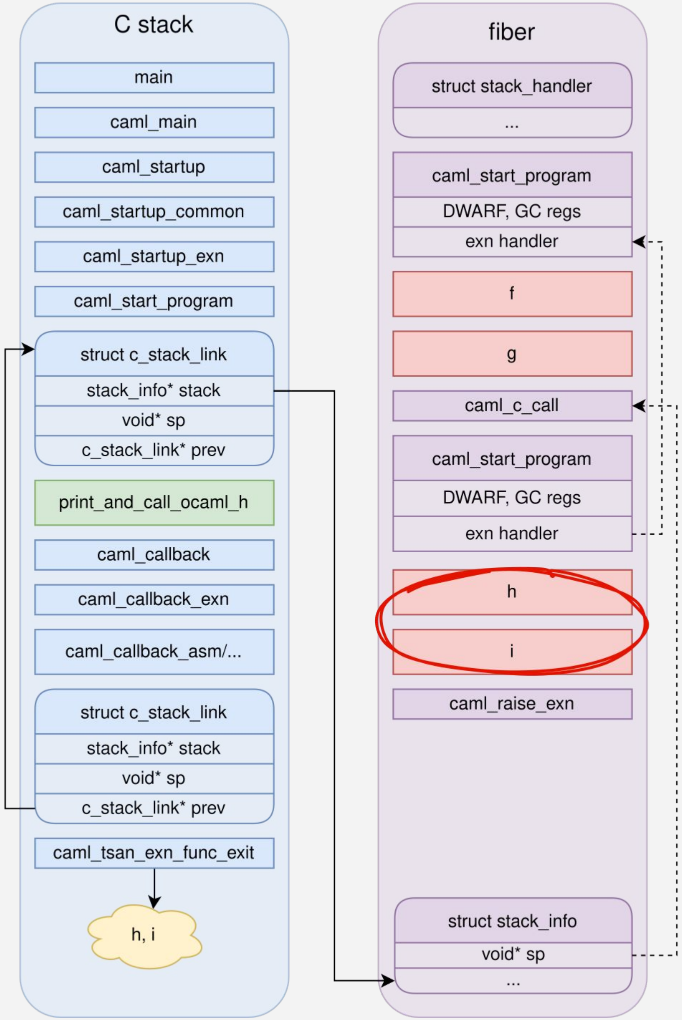


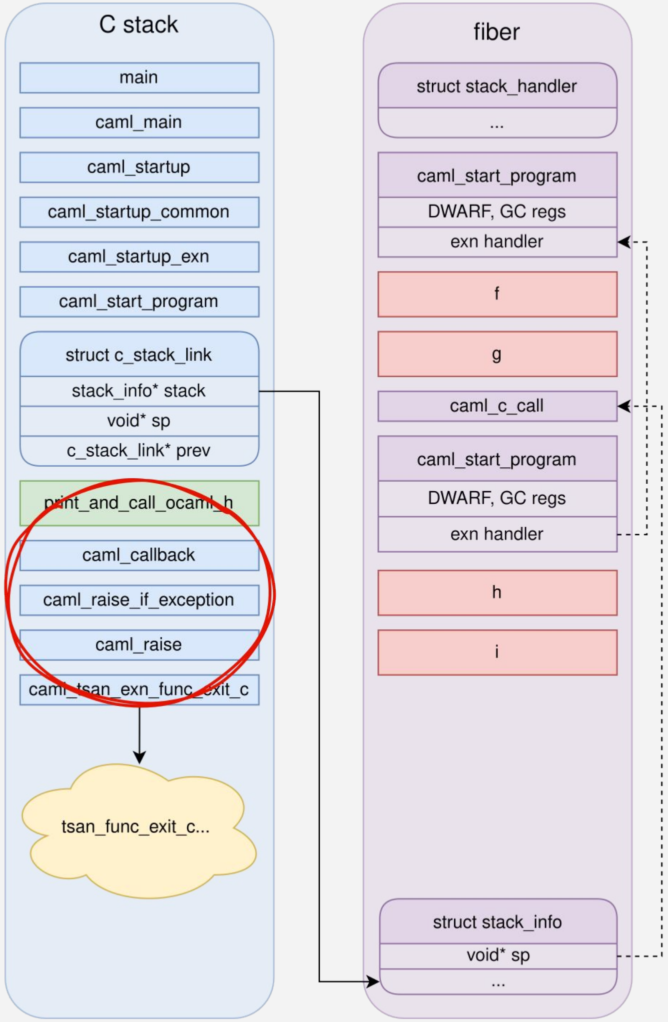
Backup slide #1: scalar clocks vs vector clocks

[Credits: go test -race Under the Hood]

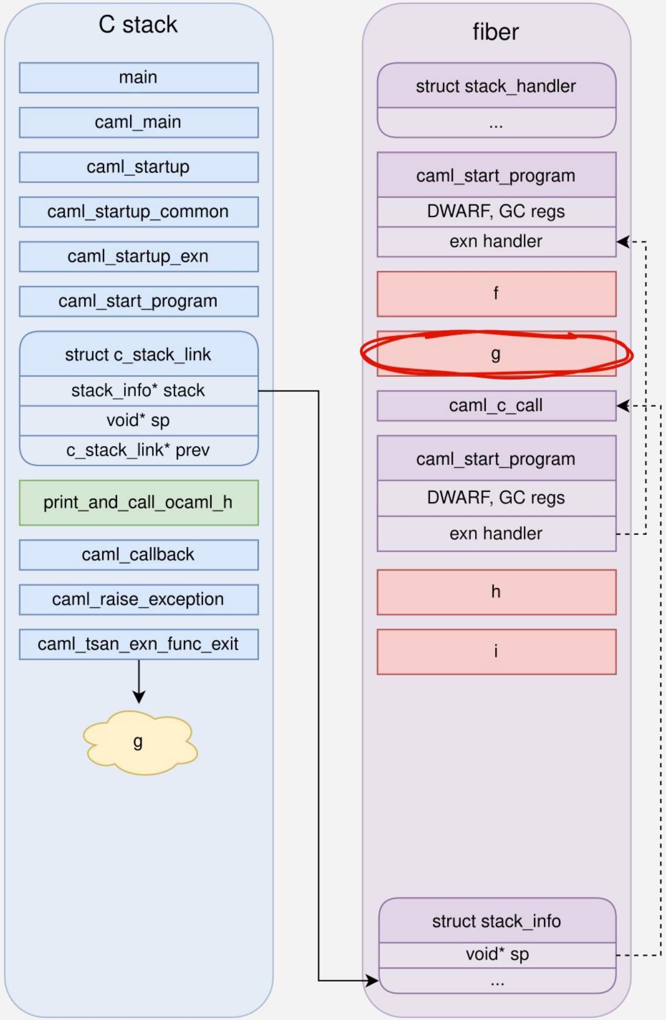


Backup slide #1: scalar clocks vs vector clocks

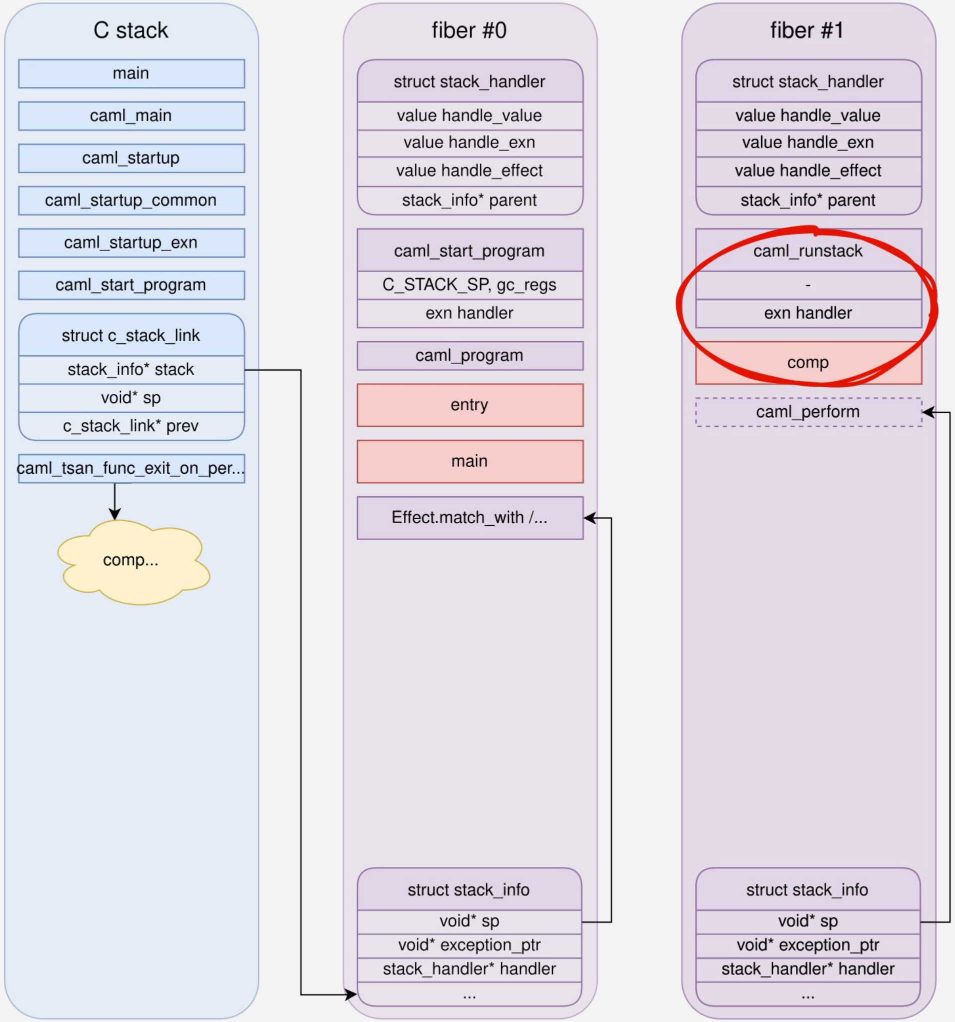

C stack
main
caml_main
caml_startup
caml_startup_common
caml_startup_exn
caml_start_program
struct c_stack_link
stack_info* stack
void* sp
c_stack_link* prev
print_and_call_ocaml_h
caml_callback
caml_callback_exn
caml_callback_asm/...
struct c_stack_link
stack_info* stack
void* sp
c_stack_link* prev
caml_tsan_exn_func_exit
h, i
fiber
struct stack_handler
...
caml_start_program
DWARF, GC regs
exn handler
f
g
caml_c_call
caml_start_program
DWARF, GC regs
exn handler
h
i
caml_raise_exn
struct stack_info
void* sp
...



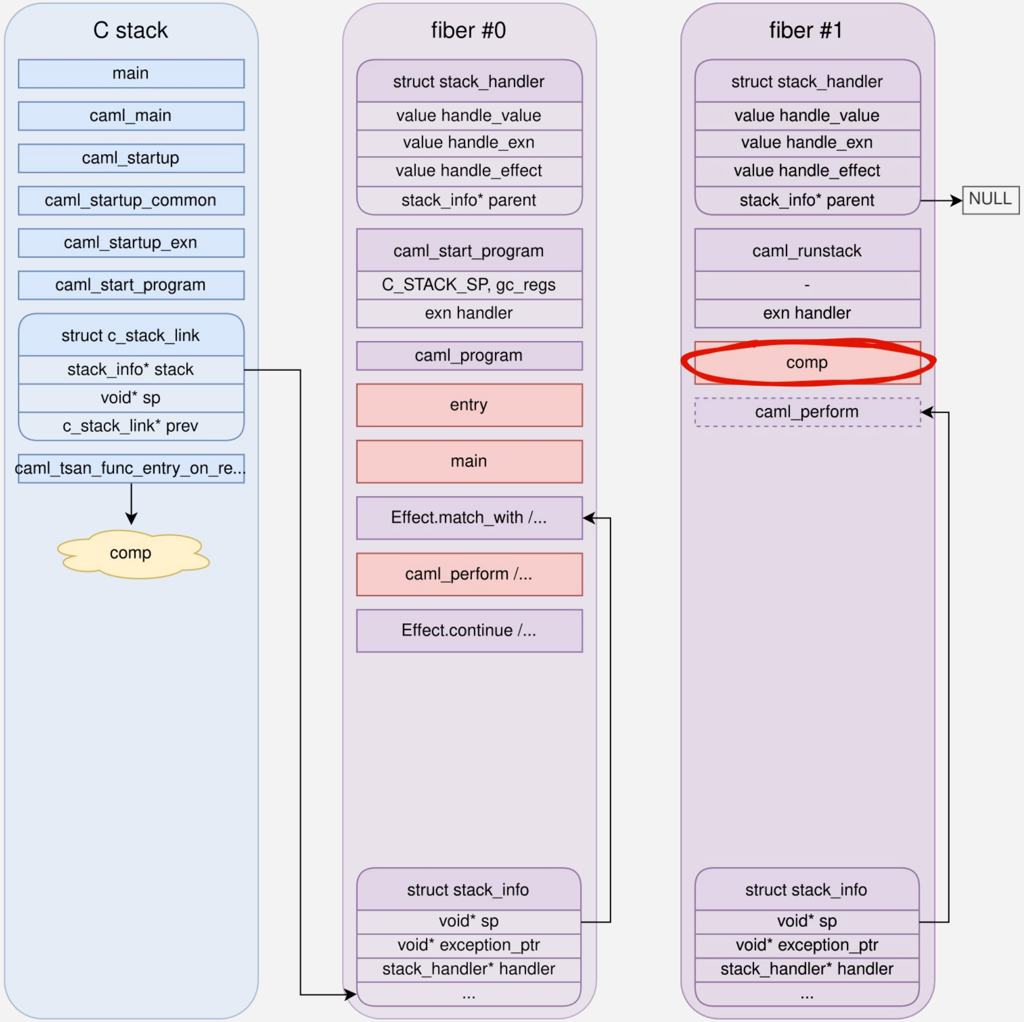
C stack
main
caml_main
caml_startup
caml_startup_common
caml_startup_exn
caml_start_program
struct c_stack_link
stack_info* stack
void* sp
c_stack_link* prev
print_and_call_ocaml_h
caml_callback
caml_raise_if_exception
caml_raise
caml_tsan_exn_func_exit_c
tsan_func_exit_c...
fiber
struct stack_handler
...
caml_start_program
DWARF, GC regs
exn handler
f
g
caml_c_call
caml_start_program
DWARF, GC regs
exn handler
h
i
struct stack_info
void* sp
...



C stack
main
caml_main
caml_startup
caml_startup_common
caml_startup_exn
caml_start_program
struct c_stack_link
stack_info* stack
void* sp
c_stack_link* prev
print_and_call_ocaml_h
caml_callback
caml_raise_exception
caml_tsan_exn_func_exit
g
fiber
struct stack_handler
...
caml_start_program
DWARF, GC regs
exn handler
f
g
caml_c_call
caml_start_program
DWARF, GC regs
exn handler
h
i
struct stack_info
void* sp
...



C stack
main
caml_main
caml_startup
caml_startup_common
caml_startup_exn
caml_start_program
struct c_stack_link
stack_info* stack
void* sp
c_stack_link* prev
caml_tsan_func_exit_on_per...
comp...
fiber #0
struct stack_handler
value handle_value
value handle_exn
value handle_effect
stack_info* parent
caml_start_program
C_STACK_SP, gc_regs
exn handler
caml_program
entry
main
Effect.match_with /...
struct stack_info
void* sp
void* exception_ptr
stack_handler* handler
...
fiber #1
struct stack_handler
value handle_value
value handle_exn
value handle_effect
stack_info* parent
caml_runstack
-
exn handler
comp
caml_perform
struct stack_info
void* sp
void* exception_ptr
stack_handler* handler
...


C stack
main
caml_main
caml_startup
caml_startup_common
caml_startup_exn
caml_start_program
struct c_stack_link
stack_info* stack
void* sp
c_stack_link* prev
caml_tsan_func_entry_on_re...
comp
fiber #0
struct stack_handler
value handle_value
value handle_exn
value handle_effect
stack_info* parent
caml_start_program
C_STACK_SP, gc_regs
exn handler
caml_program
entry
main
Effect.match_with /...
caml_perform /...
Effect.continue /...
struct stack_info
void* sp
void* exception_ptr
stack_handler* handler
...
fiber #1
struct stack_handler
value handle_value
value handle_exn
value handle_effect
stack_info* parent
NULL
caml_runstack
-
exn handler
comp
caml_perform
struct stack_info
void* sp
void* exception_ptr
stack_handler* handler
...