Effect Handlers in Low-Level Languages
Challenges and Opportunities

Programming Language Team in Edinburgh

www.huawei.com

HUAWEI

Using effect handlers from C

- **Why?**
  - Effect handlers provide: green threads, actors, generators, exceptions
  - C: only **modern** language missing **all** of these features
  - Therefore: C stands to benefit **the most**!

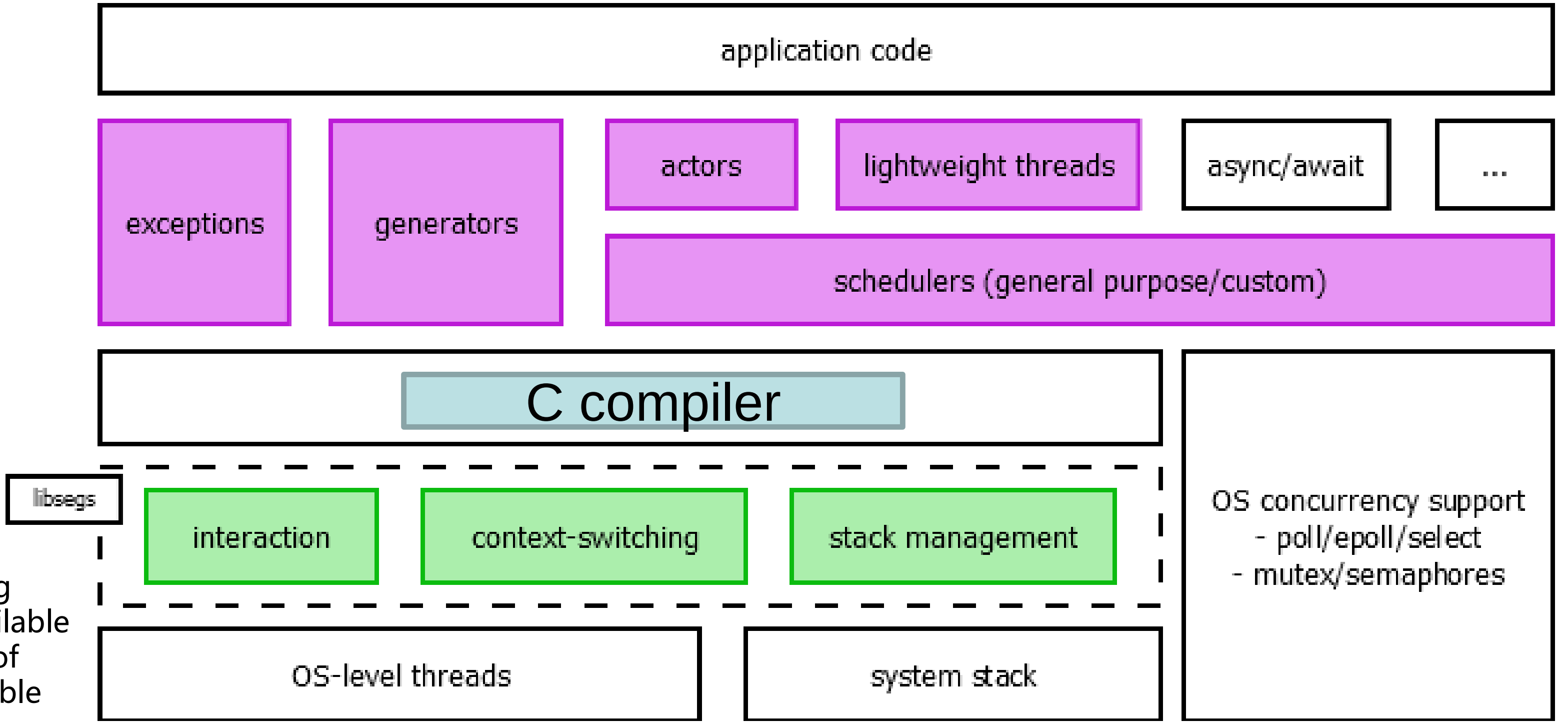
Using effect handlers from C

- **Why?**
  - Effect handlers provide: green threads, actors, generators, exceptions
  - C: only **modern** language missing **all** of these features
  - Therefore: C stands to benefit **the most**!

- **Ok, but why, really?**
  - Tons of C code in use at Huawei
  - Many projects re-invent concurrency! (Coroutines/actors built on setjmp/longjmp)
  - Main goal: use effect handlers to provide **lightweight, modular** concurrency features for C
  - Main goal: effect handlers should be **compatible with every C feature** (stack stability)
  - Main goal: effect handlers should be **ergonomic** to use **by hand**
  - Non-goal: use effect handlers to structure effectful computation
  - Non-goal (for now): statically enforce runtime safety

Stackful coroutines in C

- Offer coroutine support through `libsegs` library (currently closed-source)
- Prototype implementation in major compilers gcc & clang
- Compiler can provide **extra support, optimizations** & **better syntax**
- Small asm part needs to be ported to different architectures, **rest is architecture-independent**
- **Effects** = **stackful coroutines** + **dynamic binding** (corollary: C programmers are **not scared**)



Green: working prototype available
Purple: proof of concept available

Coroutine API

- The **coroutine** is the fundamental abstraction of **`libsegs (no resumptions/continuations)`**
- The stack frame of any function executing inside the coroutine lives in the coroutine's memory
- Once created, a coroutine may be **resumed**
- Inside a running coroutine, any function may **yield** and provide some information to the context
- Coroutines are **thread-safe** and can be sent between threads to achieve e.g. work-stealing

```
typedef struct { ... } coroutine_t;

typedef void *fun_t(coroutine_t *, void *);

coroutine_t *coroutine_new(start_fun_t *, void *);
coroutine_t *coroutine_new_sized(fun_t *, void *, size_t);
void coroutine_delete(coroutine_t *);

bool coroutine_init(coroutine_t *, fun_t *, void *);
bool coroutine_release(coroutine_t *);
```

Coroutine API

- The **coroutine** is the fundamental abstraction of **`libsegs (no resumptions/continuations)`**
- The stack frame of any function executing inside the coroutine lives in the coroutine's memory
- Once created, a coroutine may be **resumed**
- Inside a running coroutine, any function may **yield** and provide some information to the context
- Coroutines are **thread-safe** and can be sent between threads to achieve e.g. work-stealing

```
typedef struct { ... } coroutine_t;

typedef void *fun_t(coroutine_t *, void *);

coroutine_t *coroutine_new(start_fun_t *, void *);
coroutine_t *coroutine_new_sized(fun_t *, void *, size_t);
void coroutine_delete(coroutine_t *);

bool coroutine_init(coroutine_t *, fun_t *, void *);
bool coroutine_release(coroutine_t *);
```

- Coroutine & stacklet can be dynamically allocated or programmer can provide memory block
- Implementation is **untyped** (input/return is `void *`)

Coroutine API

- The **coroutine** is the fundamental abstraction of **`libsegs (no resumptions/continuations)`**
- The stack frame of any function executing inside the coroutine lives in the coroutine's memory
- Once created, a coroutine may be **resumed**
- Inside a running coroutine, any function may **yield** and provide some information to the context
- Coroutines are **thread-safe** and can be sent between threads to achieve e.g. work-stealing

```
void *segs_yield(coroutine_t *, void *);
void *segs_resume(coroutine_t *, void *);
```

Coroutine-like API (similar to e.g. libco), can yield to any "parent" coroutine (not checked)

```
typedef struct {
    effect_id id;
    void *payload;
} eff_t;
```

We use 64-bit bitsets for effects, max 64 definable effects

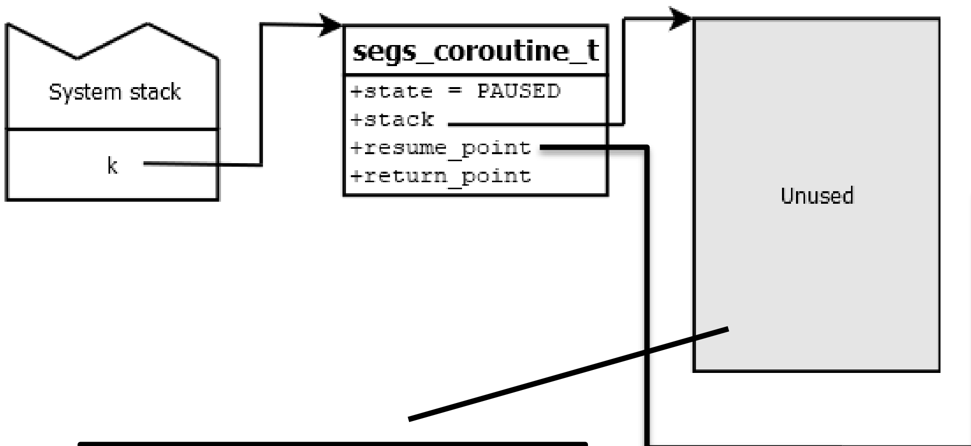
```
void *segs_perform(effect_id, void *);
eff_t *segs_handle(coroutine_t *, void *, effect_set);
```

Effect-like API: do not yield to specific coroutine, instead search active coroutine stack for installed handler

- Handlers are **shallow** (technically **sheep**) – see example later
- Helper macros `DEFINE_EFFECT`, `PERFORM`, `CASE_EFFECT` buy us *some* type-safety/convenience

Coroutine lifetime

- The **coroutine** is the fundamental abstraction of **`libsegs`**
- The stack frame of any function executing inside the coroutine lives in the coroutine's memory
- Once created, a coroutine may be **resumed**
- Inside a running coroutine, any function may **yield** and provide some information to the context
- Coroutines are **thread-safe** and can be sent between threads to achieve e.g. work-stealing



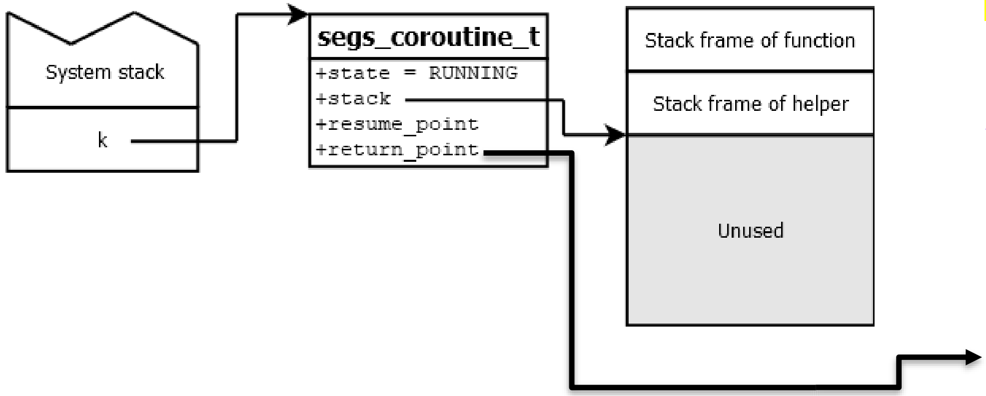
Stack segment also contains header with metadata (omitted)

```c
int64_t helper(segs_coroutine_t* self) {
    return (int64_t)segs_yield(self, (void*)10);
}

void* function(segs_coroutine_t* self, void* arg) {
    int64_t x = helper(self);
    return (void*)(x + (int64_t)arg);
}

segs_coroutine_t *k = segs_coroutine_new(function, 0);
// Before call to resume
int64_t intermediate = (int64_t)segs_resume(k, NULL);
int64_t result = (int64_t)segs_resume(k, 2 * i);
segs_coroutine_delete(k);
```

The environment can pass some data to the coroutine when resuming

Coroutine lifetime

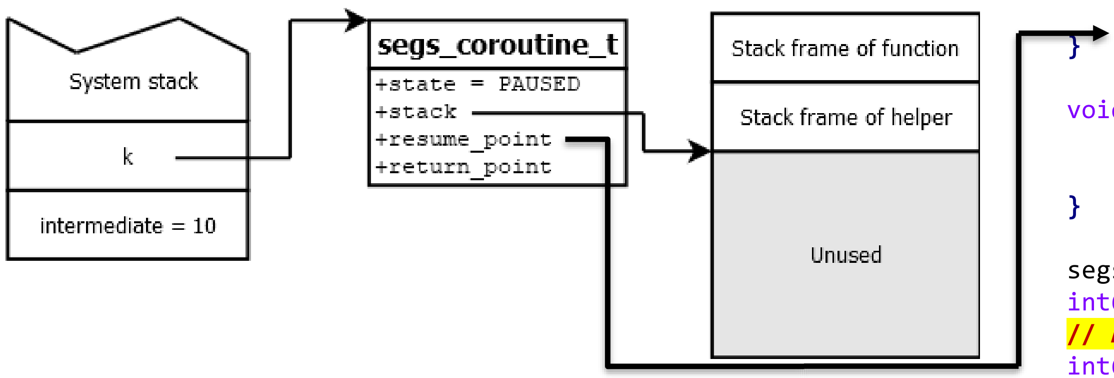
- The **coroutine** is the fundamental abstraction of **`libsegs`**
- The stack frame of any function executing inside the coroutine lives in the coroutine's memory
- Once created, a coroutine may be **resumed**
- Inside a running coroutine, any function may **yield** and provide some information to the context
- Coroutines are **thread-safe** and can be sent between threads to achieve e.g. work-stealing




```c
int64_t helper(segs_coroutine_t* self) {
    // Before call to yield
    return (int64_t)segs_yield(self, (void*)10);
}

void* function(segs_coroutine_t* self, void* arg) {
    int64_t x = helper(self);
    return (void*)(x + (int64_t)arg);
}

segs_coroutine_t *k = segs_coroutine_new(function, 0);
int64_t intermediate = (int64_t)segs_resume(k, NULL);
int64_t result = (int64_t)segs_resume(k, 2 * i);
segs_coroutine_delete(k);
```

Coroutine lifetime

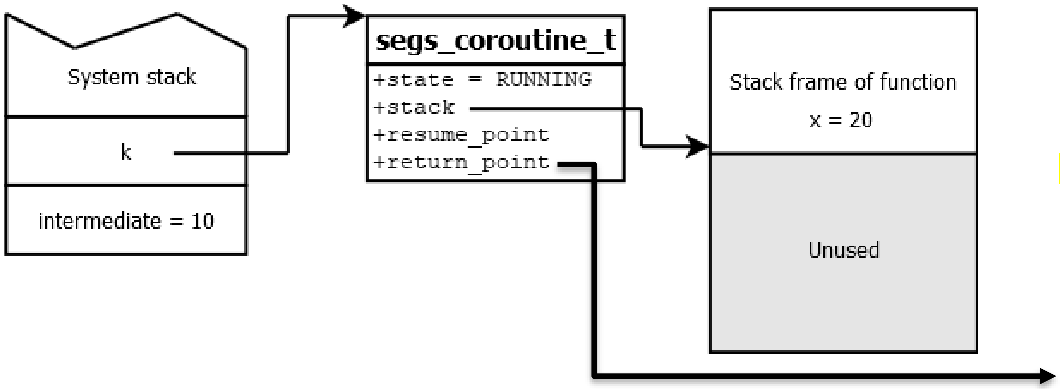
- The **coroutine** is the fundamental abstraction of **`libsegs`**
- The stack frame of any function executing inside the coroutine lives in the coroutine's memory
- Once created, a coroutine may be **resumed**
- Inside a running coroutine, any function may **yield** and provide some information to the context
- Coroutines are **thread-safe** and can be sent between threads to achieve e.g. work-stealing




```
int64_t helper(segs_coroutine_t* self) {
    return (int64_t)segs_yield(self, (void*)10);
}

void* function(segs_coroutine_t* self, void* arg) {
    int64_t x = helper(self);
    return (void*)(x + (int64_t)arg);
}

segs_coroutine_t *k = segs_coroutine_new(function, 0);
int64_t intermediate = (int64_t)segs_resume(k, NULL);
// After call to yield
int64_t result = (int64_t)segs_resume(k, 2 * i);
segs_coroutine_delete(k);
```

Coroutine lifetime

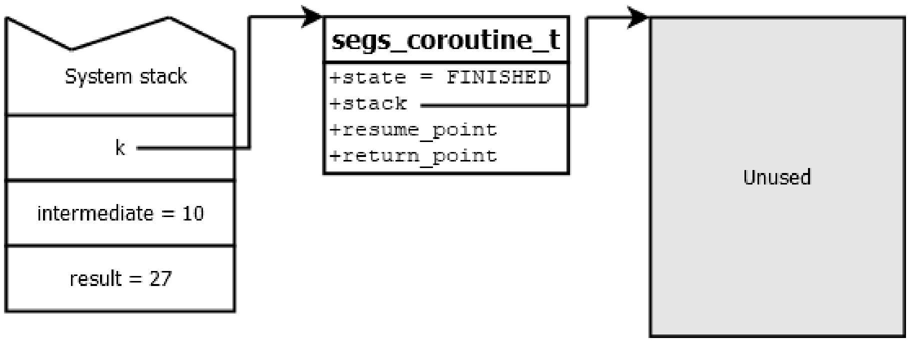
- The **coroutine** is the fundamental abstraction of **`libsegs`**
- The stack frame of any function executing inside the coroutine lives in the coroutine’s memory
- Once created, a coroutine may be **resumed**
- Inside a running coroutine, any function may **yield** and provide some information to the context
- Coroutines are **thread-safe** and can be sent between threads to achieve e.g. work-stealing




```c
int64_t helper(segs_coroutine_t* self) {
    return (int64_t)segs_yield(self, (void*)10);
}

void* function(segs_coroutine_t* self, void* arg) {
    int64_t x = helper(self);
    // After call to resume
    return (void*)(x + (int64_t)arg);
}

segs_coroutine_t *k = segs_coroutine_new(function, 0);
int64_t intermediate = (int64_t)segs_resume(k, NULL);
int64_t result = (int64_t)segs_resume(k, 2 * i);
segs_coroutine_delete(k);
```

Coroutine lifetime

- The **coroutine** is the fundamental abstraction of **`libsegs`**
- The stack frame of any function executing inside the coroutine lives in the coroutine's memory
- Once created, a coroutine may be **resumed**
- Inside a running coroutine, any function may **yield** and provide some information to the context
- Coroutines are **thread-safe** and can be sent between threads to achieve e.g. work-stealing



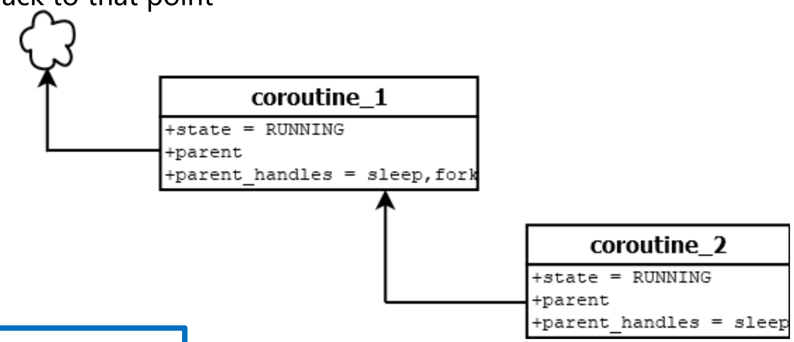

```
int64_t helper(segs_coroutine_t* self) {
    return (int64_t)segs_yield(self, (void*)10);
}

void* function(segs_coroutine_t* self, void* arg) {
    int64_t x = helper(self);
    return (void*)(x + (int64_t)arg);
}

segs_coroutine_t *k = segs_coroutine_new(function, 7);
int64_t intermediate = (int64_t)segs_resume(k, NULL);
int64_t result = (int64_t)segs_resume(k, 2 * i);
// After return
segs_coroutine_delete(k);
```

Effect example

- Increase font size in emacs is C-x C-+

Scheduler interaction

- The **coroutine** is the fundamental abstraction of **`libsegs`**
- When yielding, a coroutine can **make a request** to the scheduler that resumed it
  - E.g. it can request to fork a new coroutine, or wait until a certain device is ready
- When resuming a coroutine, the user can **provide some extra data** to fulfill the coroutine's request
- We provide some auxiliary mechanisms to automatically yield to the context that can fulfill a request
  - `Segs_handle` resumes a coroutine and promises it that certain requests can be handled by yielding back to this point
  - `PERFORM` locates the point that can handle a given request and yields back to that point




```
void* coroutine_2(segs_coroutine_t* self, void* arg) {
    PERFORM(sleep);
    PERFORM(fork, coroutine_2);
}
void* coroutine_1(segs_coroutine_t* self, void* arg) {
    ...
    k2 = segs_coroutine_new(coroutine_2, NULL);
    segs_handle(k2, NULL, HANDLES(sleep));
}
...
k1 = segs_coroutine_new(coroutine_1, NULL);
segs_handle(k1, NULL, HANDLES(sleep) | HANDLES(fork));
```

User can make requests using `PERFORM` macro

If `coroutine_2` requests `sleep`, it will yield to this point of the code

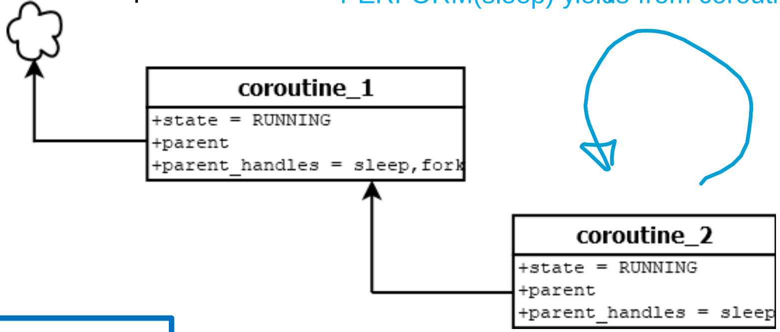
Multiple kinds of requests can be handled.
If coroutine_1 requests sleep or fork, it will yield here.
If coroutine_2 requests fork, it will also yield here

- PERFORM traverses the linked list of coroutines looking for the one that can handle the request
- Only **one** context-switch is needed, can yield directly to the handler
- Very similar to locating an exception handler
- It can be used to **implement** exceptions

Scheduler interaction

- The **coroutine** is the fundamental abstraction of **`libsegs`**
- When yielding, a coroutine can **make a request** to the scheduler that resumed it
  - E.g. it can request to fork a new coroutine, or wait until a certain device is ready
- When resuming a coroutine, the user can **provide some extra data** to fulfill the coroutine's request
- We provide some auxiliary mechanisms to automatically yield to the context that can fulfill a request
  - `Segs_handle` resumes a coroutine and promises it that certain requests can be handled by yielding back to this point
  - `PERFORM` locates the point that can handle a given request and yields back to that point

PERFORM(sleep) yields from coroutine_2




```
void* coroutine_2(segs_coroutine_t* self, void* arg) {
    PERFORM(sleep);
    PERFORM(fork, coroutine_2);
}
void* coroutine_1(segs_coroutine_t* self, void* arg) {
    ...
    k2 = segs_coroutine_new(coroutine_2, NULL);
    segs_handle(k2, NULL, HANDLES(sleep));
}
...
k1 = segs_coroutine_new(coroutine_1, NULL);
segs_handle(k1, NULL, HANDLES(sleep) | HANDLES(fork));
```

User can make requests using `PERFORM` macro

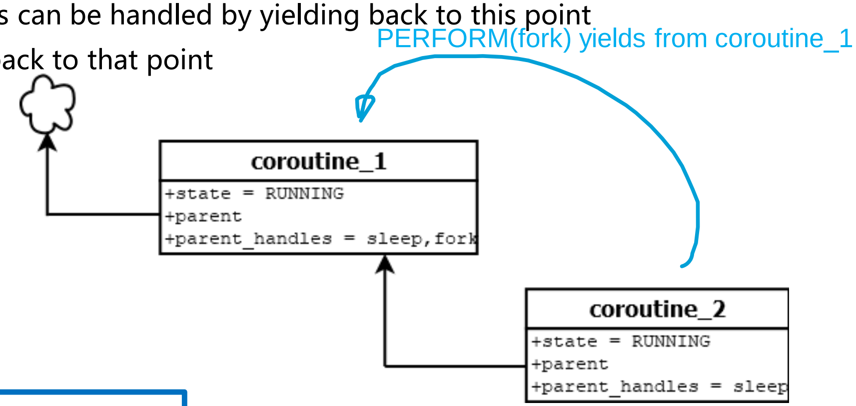
If `coroutine_2` requests `sleep`, it will yield to this point of the code

Multiple kinds of requests can be handled.
If coroutine_1 requests sleep or fork, it will yield here.
If coroutine_2 requests fork, it will also yield here

- PERFORM traverses the linked list of coroutines looking for the one that can handle the request
- Only **one** context-switch is needed, can yield directly to the handler
- Very similar to locating an exception handler
- It can be used to **implement** exceptions

Scheduler interaction

- The **coroutine** is the fundamental abstraction of **`libsegs`**
- When yielding, a coroutine can **make a request** to the scheduler that resumed it
  - E.g. it can request to fork a new coroutine, or wait until a certain device is ready
- When resuming a coroutine, the user can **provide some extra data** to fulfill the coroutine's request
- We provide some auxiliary mechanisms to automatically yield to the context that can fulfill a request
  - `Segs_handle` resumes a coroutine and promises it that certain requests can be handled by yielding back to this point
  - `PERFORM` locates the point that can handle a given request and yields back to that point




```
void* coroutine_2(segs_coroutine_t* self, void* arg) {
    PERFORM(sleep);
    PERFORM(fork, coroutine_2);
}
void* coroutine_1(segs_coroutine_t* self, void* arg) {
    ...
    k2 = segs_coroutine_new(coroutine_2, NULL);
    segs_handle(k2, NULL, HANDLES(sleep));
}
...
k1 = segs_coroutine_new(coroutine_1, NULL);
segs_handle(k1, NULL, HANDLES(sleep) | HANDLES(fork));
```

User can make requests using `PERFORM` macro

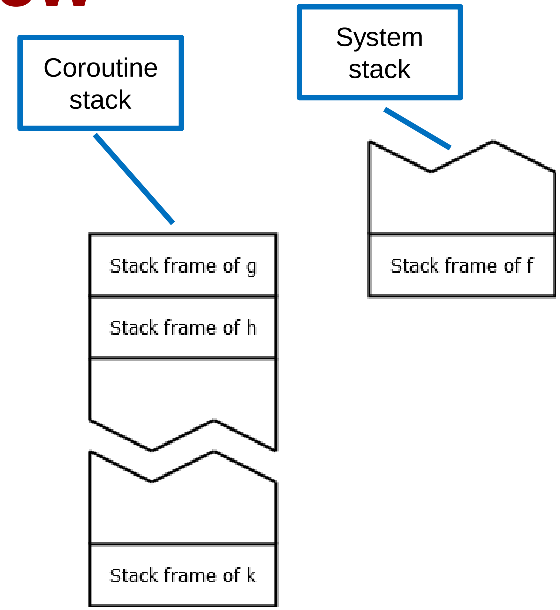
If `coroutine_2` requests `sleep`, it will yield to this point of the code

Multiple kinds of requests can be handled.
If coroutine_1 requests sleep or fork, it will yield here.
If coroutine_2 requests fork, it will also yield here

- PERFORM traverses the linked list of coroutines looking for the one that can handle the request
- Only **one** context-switch is needed, can yield directly to the handler
- Very similar to locating an exception handler
- It can be used to **implement** exceptions

Stackful coroutines: an overview

- Commonly implemented with runtime support
  - Lua (coroutines, built-in)
  - Go (goroutines, built-in)
  - Java (virtual threads, built-in since Java 19)
  - C++ (Boost::Coroutine, implemented as a library)
  - Rust (may, implemented as a library)
  - Erlang (processes, built-in)
- Allocate **entire stack** (not just one frame) for coroutine
  - Stack space can be allocated in heap, global memory, or anywhere
- All calls inside coroutine use coroutine stack
- **Any** function within the coroutine may yield
- Can use **static-sized** stacks or **growable** stacks
  - Growable stacks need more runtime support
- No difference between sync/async functions
  - All functions can call async functions

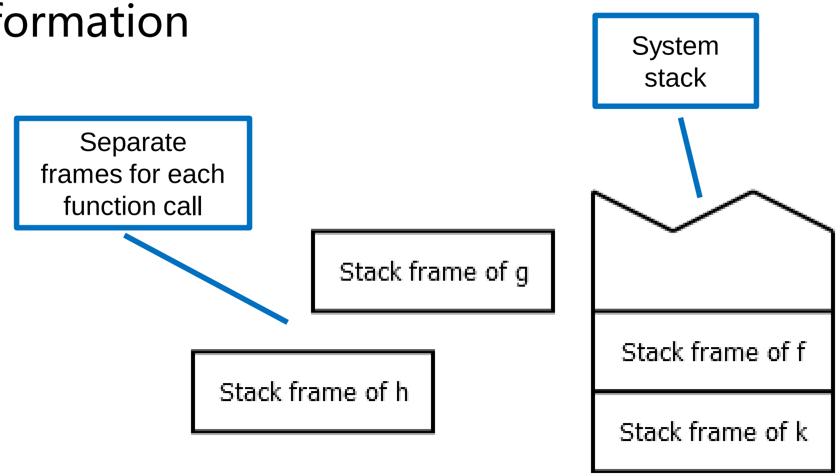


No distinction between sync/async

```
int k() { yield; return 1; }
int h() { yield; return k(); }
int g() { h(); }

void f() {
    coroutine* coro = create_coroutine(g);
}
```

Stackless concurrency: an overview

- Commonly implemented via compiler transformation
  - C++ (C++20 coroutines/libcoro)
  - Rust
  - Kotlin
  - Swift
  - Javascript
- Create **single stack frame** for coroutine
  - Frame can be allocated anywhere
  - Function is transformed into state machine
- Calls inside coroutine use system stack
- Can only yield from top-level function
  - Can yield from nested coroutine with special **`await`** syntax
  - Without complex optimizations, nesting coroutines can be very expensive! (one allocation per coroutine call, chaining yields...)
- Async functions are **special**
  - E.g. cannot be used as function pointers



Can only use **`yield`** in **`async`** functions

```
int k() { yield; return 1; }
int h() async { yield; return k(); }
int g() async { await h(); }

void f() {
    coroutine* coro = g();
}
```

Hypothetical syntax for stackless coroutines in C

- `yield` *for pausing the current coroutine*
- `await` *for nesting coroutine calls*
- `async` *for marking coroutine functions*

Stackful vs Stackless

- Stackful offers **better modularity**
- With stackless: making function async/not async means changing all callers!
- Example: adding async logging code to computation

```
void log(string s) async { ... }
int complex_computation(int[] inputs) {
    ...
    if (error) throw "error!";
}
int test() async {
    int x = complex_computation(args);
    ...
}
int main() {
    int x = complex_computation(args);
    ...
}
```

Stackless coroutines

```
void log(string s) async { ... }
int complex_computation(int[] inputs) async {
    ...
    if (error) await log("error!");
}
int test() async {
    int x = await complex_computation(args);
    ...
}
int main() {
    int x = run_task(complex_computation(args));
    ...
}
```

Need to change callers

Different changes depending if caller is async

```
void log(string s) { ... }
int complex_computation(int[] inputs) {
    ...
    if (error) throw "error!";
}
int test() {
    int x = complex_computation(args);
    ...
}
int main() {
    int x = complex_computation(args);
    ...
}
```

Stackful coroutines

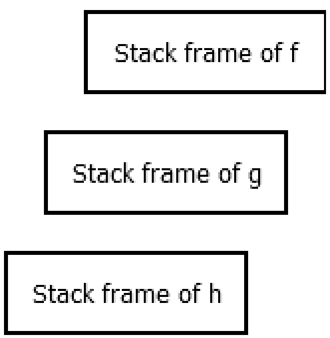
All code remains exactly the same!

```
void log(string s) { ... }
int complex_computation(int[] inputs) {
    ...
    if (error) log("error!");
}
int test() {
    int x = complex_computation(args);
    ...
}
int main() {
    int x = complex_computation(args);
    ...
}
```

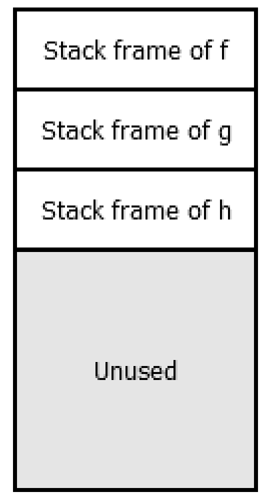
Stackful challenges

- Needs architecture-specific support (portable C library, binds to small platform-dependent asm)
- More complex stack management
  - Resizable stacks
    - Virtual mem
    - Stack copying — Not suitable for low-level!
    - Segmented stacks — Complex, some runtime overhead
  - Fixed-size stacks — Some memory waste, no recursion
- Cost of context switch — 20~30 μinstructions
- **Less efficient use of memory**

Stackless:
- Each frame is allocated independently
- More allocations
- But less memory usage!




```
int h() async {
  yield; return 1;
}
int g() async {
  yield; await h();
}
int f() async {
  await g();
}
```

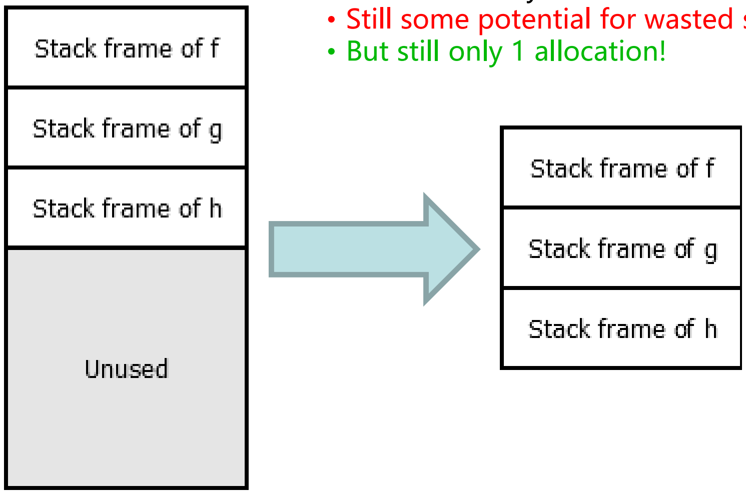


Stackful:
- All frames stored in a single memory block
- Some wasted space
- But only 1 allocation!

Stackful challenges

- Needs architecture-specific support (portable C library, binds to small platform-dependent asm)
- More complex stack management
- Cost of context switch
- **Less efficient use of memory**
  - Many optimizations are possible for stackful

```
int h() {
    yield;
    return 1;
}
int g() {
    yield;
    h();
}
int f() {
    g();
}
```



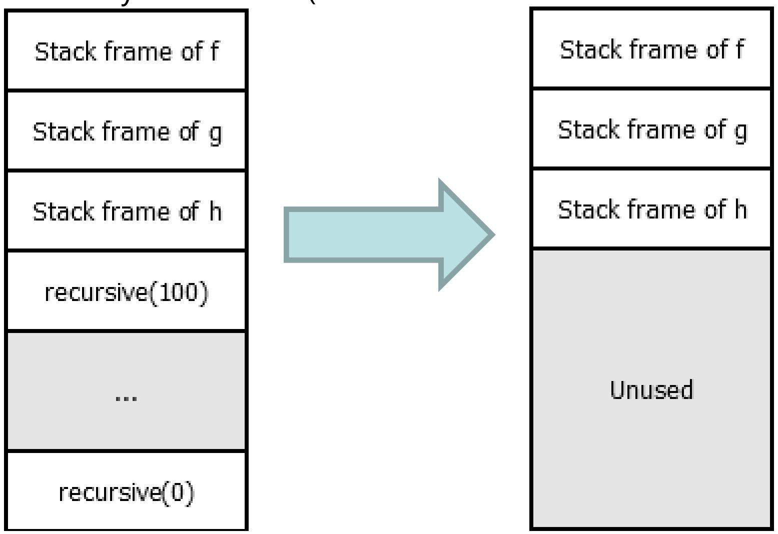
- When compiler can determine max stack frame size, can allocate exactly what is needed!
- Still some potential for wasted space
- But still only 1 allocation!

Stackful challenges

- Needs architecture-specific support (portable C library, binds to small platform-dependent asm)
- More complex stack management
- Cost of context switch
- **Less efficient use of memory**
  - Many optimizations are possible for stackful

- If compiler can determine max stack frame size, can allocate exactly what is needed!
- **Still some potential for wasted space**
  - After recursive function ends, stack frames are removed but memory cannot be easily deallocated! (Would need to reallocate stack frames of `f`, g, h)

```
int rec(int n) {
    ... rec(n-1);
}
int h() {
    yield;
    recursive(100);
    yield;
}
int g() {
    yield; h();
}
int f() {
    yield; g();
}
```



- Need to allocate stack space for recursive call
- After recursive call, space is not freed, but will not be used!

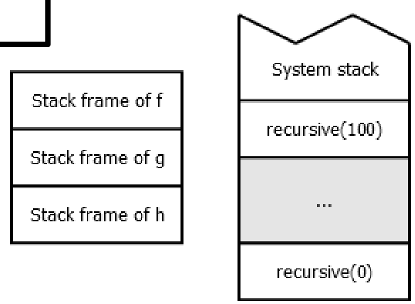
Stackful challenges

- Needs architecture-specific support (portable C library, binds to small platform-dependent asm)
- More complex stack management
- Cost of context switch
- **Less efficient use of memory**
  - Many optimizations are possible for stackful

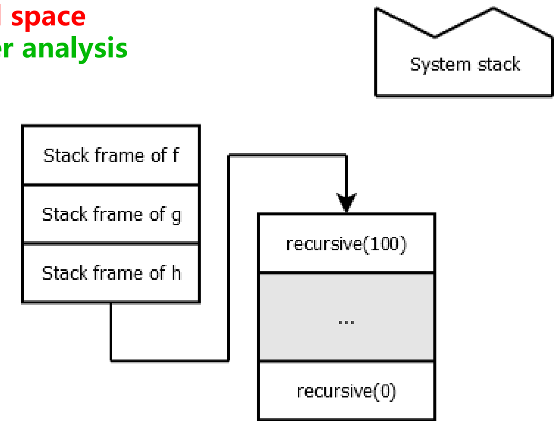
```
int recur(int n) {
    ... recur(n-1);
}
int h() {
    yield;
    recursive(100);
    yield;
}
int g() {
    yield; h();
}
int f() {
    yield; g();
}
```

No **yield** → can run on system stack

- If compiler can determine max stack frame size, can allocate exactly what is needed!
- **Still some potential for wasted space**
- **Can be mitigated with compiler analysis**



Run `recur` on the system stack
- No need for a large coroutine stack!
- But depends on compiler analysis



Use segmented stack
- Separate segments for different function calls
- Deallocate/reuse when finished
- Some runtime penalty

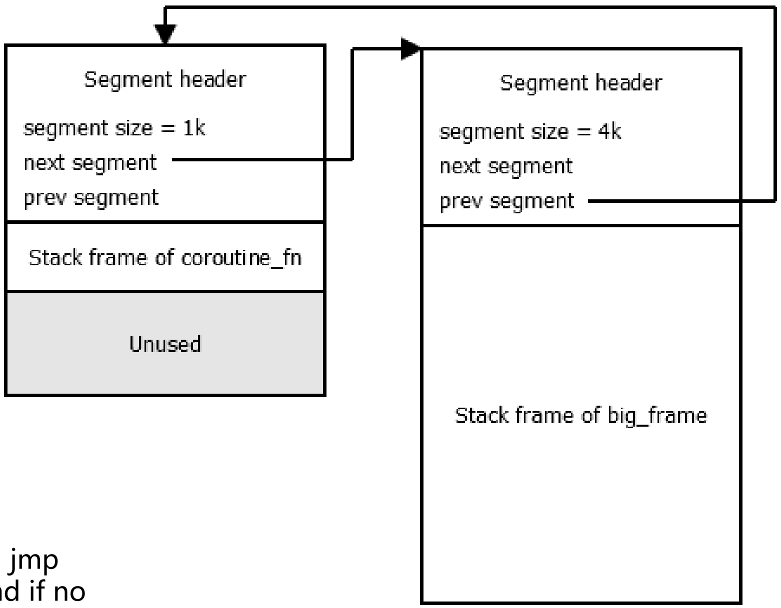
- Both approaches can be combined
- **Use compiler analysis to decide strategy**
- **If call tree is known at compile time, memory usage can be optimal**

Stack handling

- `libsegs` uses **segmented stacks** for stack handling
  - But can easily be adapted to **stack copying** or **virtual memory** if the architecture supports it!
- Coroutines are given an **initial stack** (size can be chosen by the programmer)
- Every function call **checks available stack space** vs **function stack frame size**
  - If not enough available, **new segment is allocated**
  - The check and allocation are inserted automatically by compiler (clang & gcc –fsplit–stack support)

```
int big_frame() {
    int array[1024];
    ...
    return 0;
}
int coroutine_fn() {
    big_frame();
}
```

- The current stack segment is too small for the big array
- `big_frame` checks stack size in **function prelude**, creates new stack segment in doubly-linked list




```
big_frame():
        lea     r11, [rsp - 4104]
        cmp     r11, qword ptr fs:[112]
        ja      .LBB0_0
        mov     r10d, 4104
        mov     r11d, 0
        call    __morestack
        ret
.LBB0_0:
        …
        ret
```

Prelude: check stack size & allocate

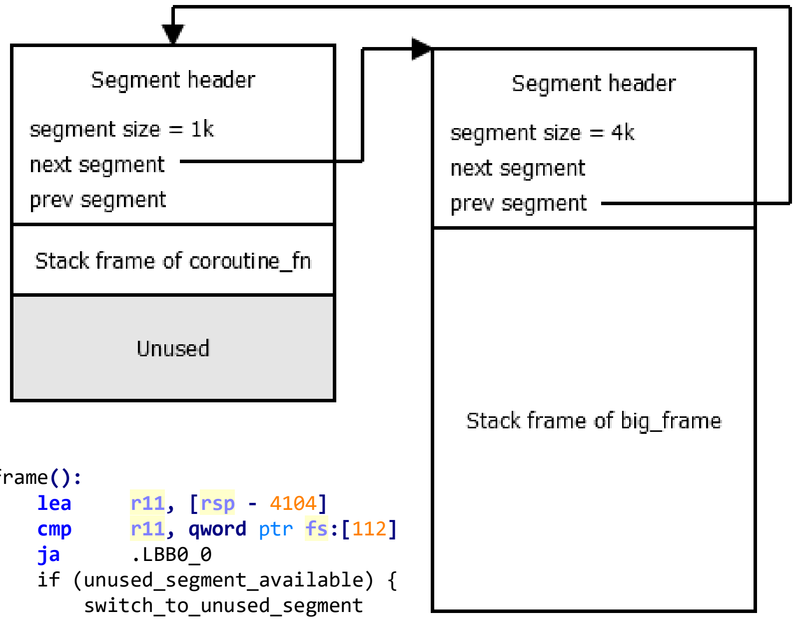
- Prelude is very cheap: load + cmp + jmp
- Branch predictor eliminates overhead if no allocation is needed
- Slow path only taken when stack needs resizing

Stack handling

- `libsegs` uses **segmented stacks** for stack handling
- Coroutines are given an **initial stack** (size can be chosen by the programmer)
- Every function call **checks available stack space** vs **function stack frame size**
- Potential performance issue: **hot split problem**

```
int big_frame() {
    int array[1024];
    ...
    return 0;
}
int coroutine_fn() {
    for (int i = 0; i < 1000000; i++) {
        big_frame();
    }
}
```

- `big_frame` allocates new segment, but is deleted at end of function call
- Allocate and deallocate 1M segments?!




```
big_frame():
        lea     r11, [rsp - 4104]
        cmp     r11, qword ptr fs:[112]
        ja      .LBB0_0
        if (unused_segment_available) {
            switch_to_unused_segment
            ja      .LBB0_0
        }
        mov     r10d, 4104
        mov     r11d, 0
        call    __morestack
        ret
.LBB0_0:
        …
        ret
```

Can be solved with runtime support!

- Do not deallocate segment upon return, just change pointers
- No need to allocate new segment, just reuse old segment!
- Allocation is replaced by just switching stack ptr
- Change autogenerated function prelude to do check: minimal overhead

Can be solved with compiler analysis!

- Detect big allocation in `big_frame`, lift it to `coroutine_fn`
- Effectively: combine stack frames of `big_frame` and `coroutine_fn`

Benchmarks: context-switching

- We compare **libsegs**, libco (Tencent's stackful coroutine library) and C++ coroutines (with cppcoro)
  - **libco** is used in **real-world applications** (currently in WeChat backend!)
- All benchmarks running on clang 10.0.0 at optimization level 3
- Context-switching: create a coroutine and resume/yield n times
- We control three different variables: number of yield/resume, depth of the stack, and size of each stack frame

```
void *deep_coroutine(segs_coroutine_t *self, void *arg) {
    char arr[padding];
    int64_t depth = (int64_t)arg;
    if (depth == 0) {
        volatile bool loop = true;
        while (loop) {
            segs_yield(self, nullptr);
        }
        return arr;
    } else {
        deep_coroutine(self, (void *)(depth - 1));
        return arr;
    }
}
```

Frames padded with uninitialized data

Infinite loop, volatile to avoid optimizations

No tail call to avoid optimizations

libsegs version

```
segs_coroutine_t *k1 = segs_coroutine_new(fn, (void *)depth);
segs_coroutine_t *k2 = segs_coroutine_new(fn, (void *)depth);
for (size_t i = 0; i < iterations / 2; i++) {
    segs_resume(k1, nullptr);
    segs_resume(k2, nullptr);
}
segs_coroutine_delete(k1);
segs_coroutine_delete(k2);
```

Driver code interleaves execution of 2 coroutines `iterations/2` times each

Benchmarks: context-switching

- We compare **`libsegs`**, **`libco`** (Tencent's stackful coroutine library) and C++ coroutines (with **`cppcoro`**)
- All benchmarks running on clang 10.0.0 at optimization level 3
- Context-switching: create a coroutine and resume/yield n times
- We control three different variables: number of yield/resume, depth of the stack, and size of each stack frame

```
void *deep_coroutine(void *arg) {
    char arr[padding];
    int64_t depth = (int64_t)arg;
    if (depth == 0) {
        volatile bool loop = true;
        while (loop) {
            co_yield_ct();
        }
        return arr;
    } else {
        deep_coroutine((void *)(depth - 1));
        return arr;
    }
}
```

Frames padded with uninitialized data

Infinite loop, volatile to avoid optimizations

No tail call to avoid optimizations

`libco` version

API is almost identical to **`libsegs`**

```
stCoRoutine_t *k1;
stCoRoutine_t *k2;
stShareStack_t *share_stack
    = co_alloc_sharestack(1, 1024 * 128);
stCoRoutineAttr_t attr;
attr.stack_size = 0;
attr.share_stack = share_stack;
co_create(&k1, &attr, fn, (void *)depth);
co_create(&k2, &attr, fn, (void *)depth);
for (size_t i = 0; i < iterations / 2; i++) {
    co_resume(k1);
    co_resume(k2);
}
co_release(k1);
co_release(k2);
```

Set share_stack on (for resizable coroutines, otherwise stack size is fixed)

Benchmarks: context-switching

- We compare **`libsegs`**, **`libco`** (Tencent's stackful coroutine library) and C++ coroutines (with **`cppcoro`**)
- All benchmarks running on clang 10.0.0 at optimization level 3
- Context-switching: create a coroutine and resume/yield n times
- We control three different variables: number of yield/resume, depth of the stack, and size of each stack frame

`cppcoro` version

`cppcoro` api does not allow for an exact comparison. We use `recursive_generator` here because it is more optimized, but **`cppcoro`** recursive generators are more limited than coroutines (no async).

```
cppcoro::recursive_generator<char *>
deep_coroutine(int64_t depth) {
    char arr[padding];
    if (depth == 0) {
        volatile bool loop = true;
        while (loop) {
            co_yield arr;
        }
    } else {
        co_yield deep_coroutine_rec (depth - 1);
        co_yield arr;
    }
}
```

Frames padded with uninitialized data

Infinite loop, volatile to avoid optimizations

No tail call to avoid optimizations

```
cppcoro::recursive_generator<char*> k1
    = coroutine_fn(depth);
cppcoro::recursive_generator<char*> k2
    = coroutine_fn(depth);
cppcoro::recursive_generator<char*>
    ::iterator k1_iter = k1.begin();
cppcoro::recursive_generator<char*>
    ::iterator k2_iter = k2.begin();
for (auto i = 0; i < iterations / 2; i++) {
    k1_iter++;
    k2_iter++;
}
```

`cppcoro` coroutines are heap-allocated, but RAII so there is no explicit deallocation

Benchmarks: context-switching

- We compare **`libsegs`**, **`libco`** (Tencent's stackful coroutine library) and C++ coroutines (with **`cppcoro`**)
- All benchmarks running on clang 10.0.0 at optimization level 3
- Context-switching: create a coroutine and resume/yield n times
- We control three different variables: number of yield/resume, depth of the stack, and size of each stack frame

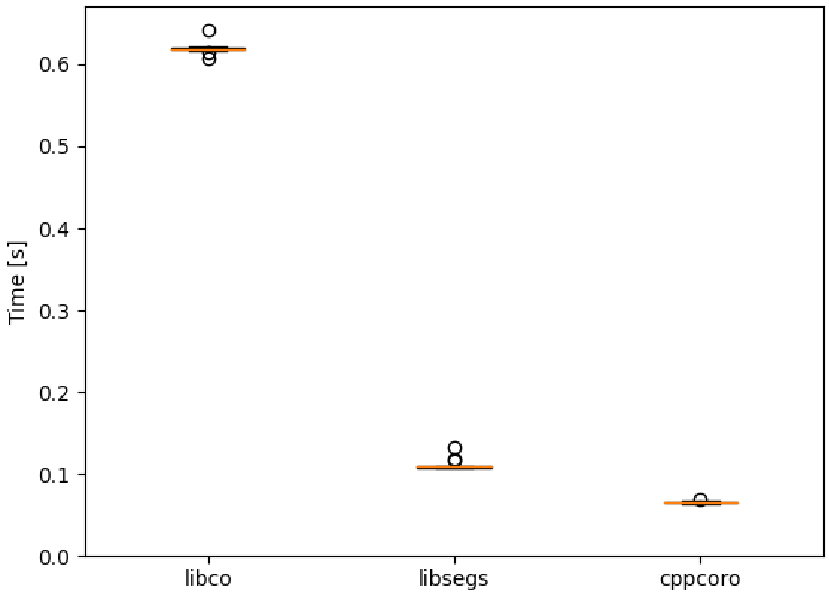
If –O0 we're actually 4.3x FASTER than cppcoro!

Simple example

- 10,000,000 iterations (resume + yield)
- Call stack has depth 0
- Stack frames have no padding

Framework	Mean time (ms)	Relative
`libco`	619.9 ± 7.3	9.42 ± 0.20
`libsegs`	110.1 ± 5.4	1.67 ± 0.09
`cppcoro`	65.8 ± 1.2	1.00

OUTDATED! We're a bit better now (~88ms)



Conclusions

- **`libsegs`** is much more efficient than **`libco`**, due to using split stacks instead of stack copying
- **`cppcoro`** is faster, but less flexible (benchmark code could not be extended with async)

Benchmarks: context-switching

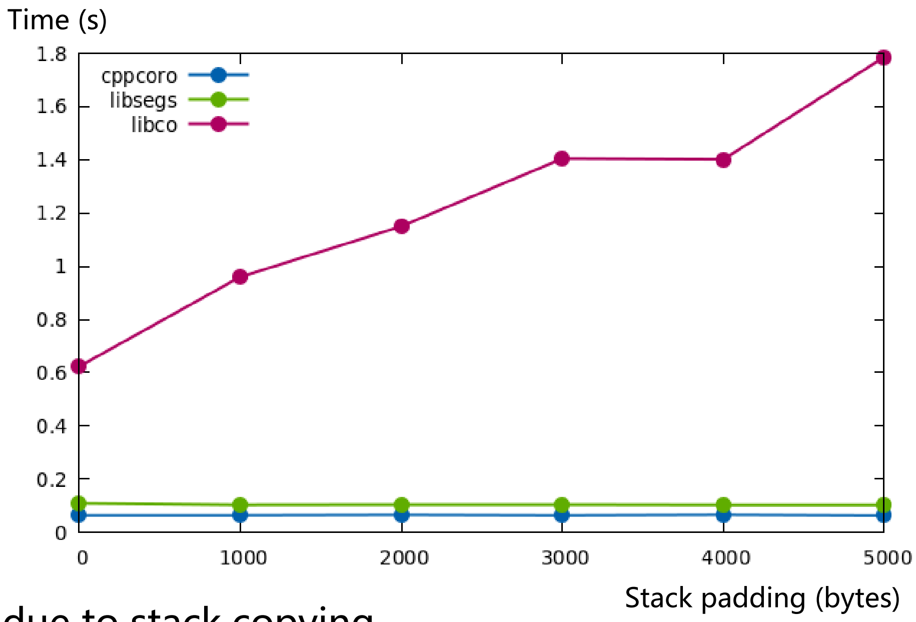
- We compare **`libsegs`**, **`libco`** (Tencent's stackful coroutine library) and C++ coroutines (with **`cppcoro`**)
- All benchmarks running on clang 10.0.0 at optimization level 3
- Context-switching: create a coroutine and resume/yield n times
- We control three different variables: number of yield/resume, depth of the stack, and size of each stack frame

Stack size scaling

- 10,000,000 iterations (resume + yield)
- Call stack has depth 0
- Stack frames have 0-5kb of padding

Size = 5000

Framework	Mean time (ms)	Relative
`libco`	1,786.1 ± 7.9	28.09 ± 1.19
`libsegs`	102.2 ± 2.8	1.61 ± 0.08
`cppcoro`	63.6 ± 2.7	1.00



Conclusions

- As expected, **`libco`** scales linearly with stack size due to stack copying
- Performance of **`libsegs`** and **`cppcoro`** is independent of stack size

Benchmarks: memory usage

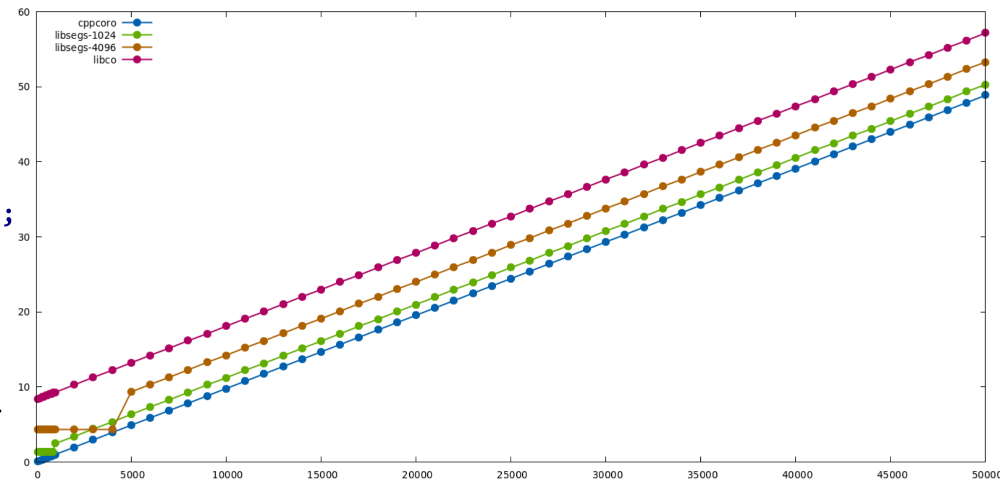
- We compare **`libsegs`**, **`libco`** (Tencent's stackful coroutine library) and C++ coroutines (with **`cppcoro`**)
- All benchmarks running on clang 10.0.0 at optimization level 3
- Memory usage: create 10k coroutines with M bytes of stack padding, then immediately pause all of them

```
cppcoro::recursive_generator<char *> *coroutines;
coroutines = new cppcoro::recursive_generator<char *>[instances];
for (auto i = 0; i < instances; i++) {
    coroutines[i] = coroutine_fn(depth);
    auto k_iter = coroutines[i].begin();
    k_iter++;
}
```

All coroutines are started to ensure memory is actually allocated

```
segs_coroutine_t **coroutines;
coroutines = new segs_coroutine_t *[instances];
for (auto i = 0; i < instances; i++) {
    coroutines[i] = segs_coroutine_new(fn, (void *)depth);
    segs_resume(coroutines[i], nullptr);
}
```

Total memory consumption (kb) vs stack frame size



Warning

- **This is not a realistic benchmark**. In reality, the memory consumption will be very different depending on compiler optimizations, depth of the call stack and the structure of the program itself!

Conclusions

- We measure **absolute** memory consumption vs the size of the stack frame for all three frameworks
  - **`cppcoro`** always allocates exactly enough memory to contain the size of the frame and no more
  - **`libsegs`** allocates an initial segment of a fixed size. If the stack frame exceeds that size, additional space is allocated to make up for the difference. There is a small fixed penalty over **`cppcoro`** due to overhead
  - **`libco`** behaves similarly to **`libsegs`**, however it incurs more initial overhead as it allocates a large initial arena

Case study

- Goal 1: showcase performance of **`libsegs`** features in "realistic" application
- Goal 2: show how to write applications and schedulers using **`libsegs`**
  1. "Proof of concept" multi-threaded scheduler with async capabilities based on epoll (can easily be adapted to poll/select)
  2. Echo server built on example scheduler, using "listen-accept-fork" approach with coroutines
  3. Benchmark single-threaded performance & multi-threaded scaling
  4. Compare against plain C event-loop style implementation

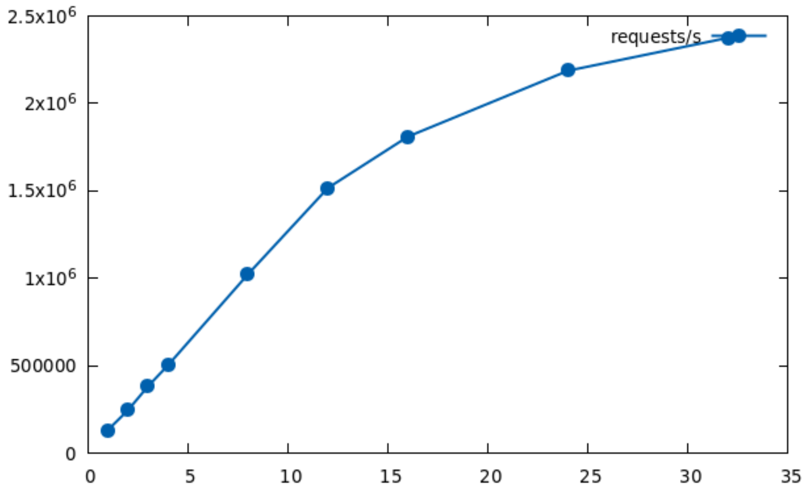
Single-threded performance

- Competitive with plain C implementation, despite using heap allocated coroutines and thread-safe queue!
- Performance degrades slightly with number of concurrent connections (caused by extra synchronization overhead but no real parallelism)
- Shows that **coroutines do not introduce significant overhead**

Impl	Requests/s (100 connections)	Requests/s (500 connections)
`libsegs`	~134k	~132k
`plain c`	~136k	~136k

Multi-threaded performance

- Linear scaling up to ~12 cores, diminishing returns above that
- Somewhat unrealistic due to use of fixed-size task queues (would segfault on overflow)
- **Implementation is very naïve**, likely can be much more efficient by experts



Conclusions

- **Enormous potential for effects in C**
  - Can be ergonomic & efficient **without** compiler support!
  - But **lots of low-hanging fruit** for compiler support
    - Type-safety, optimizations
- **Major pain point: segmented stacks**
  - **No real alternative:** virtual memory/stack copying **unworkable**
  - Opportunities for optimization
  - Gets better with proper effect typing/"purity" tracking!
- **API differences from high-level languages**
  - No try/handle blocks, continuations not exposed, coroutines as only visible abstraction
  - **Session types** obvious candidate for typing coroutines, add extra safety
- **It is worth doing!**
  - Massive gains in programmer productivity even from a minimal prototype
  - Few sharp edges, usable by non-experts!

Thank You

www.huawei.com