Developing high-performance Coroutines for ARMs

Klemens Morgenstern

10.03.2018

Introduction

- Electrical engineer by training
- C++ developer by passion
- boost contributor
- Independent contractor
- Open for consulting

Motivation

- Coroutines are awesome

Motivation

- Coroutines are awesome
- Low overhead alternative to threads

Motivation

- Coroutines are awesome
- Low overhead alternative to threads
- Do not require an OS / Scheduler

Motivation

- Coroutines are awesome
- Low overhead alternative to threads
- Do not require an OS / Scheduler
- More readable alternative to state-machines

Motivation

- Coroutines are awesome
- Low overhead alternative to threads
- Do not require an OS / Scheduler
- More readable alternative to state-machines
- Not well enough known

Overview

1. Introduction to coroutines

Overview

1. Introduction to coroutines
2. Implementation

Overview

1. Introduction to coroutines
2. Implementation
3. Performance comparisons

Definition

- Function with it's own stack

Definition

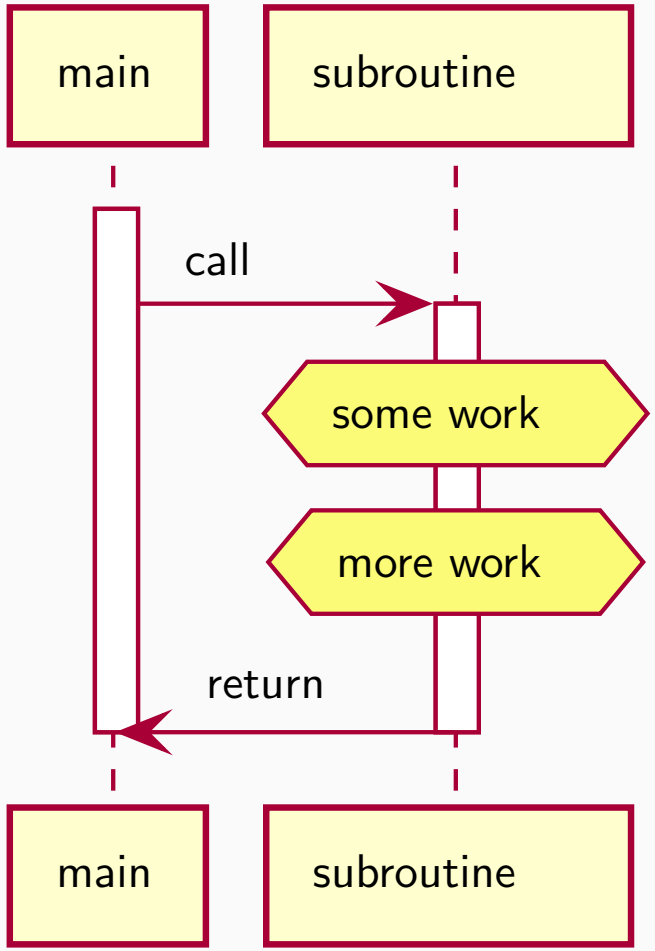
- Function with it's own stack
- can interrupt it's own execution

Definition

- Function with it's own stack
- can interrupt it's own execution
- and be resumed later

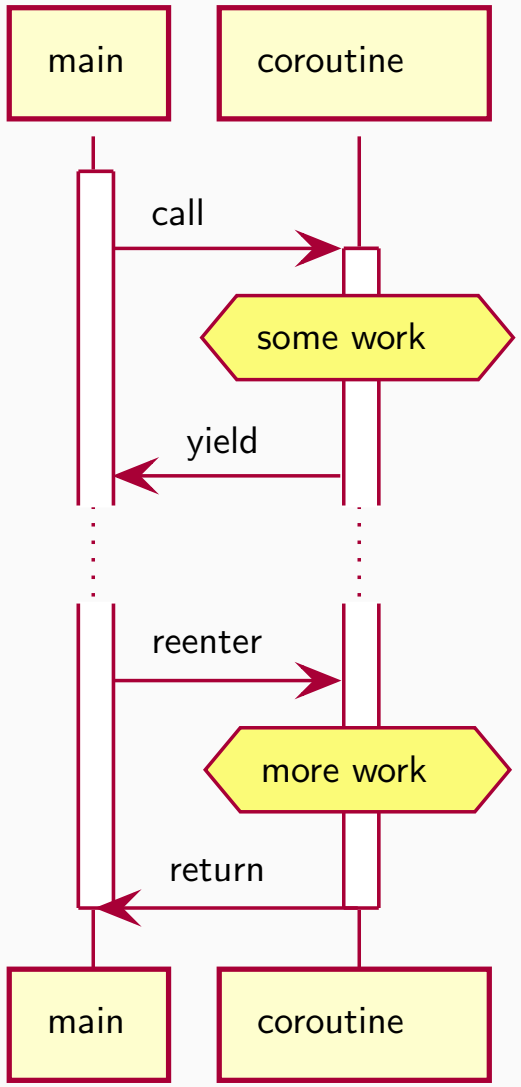
Subroutine

```
1  int main()
2  {
3    subroutine();
4    return 0;
5  }
6
7  void subroutine()
8  {
9    some_work();
10   more_work();
11 }
```



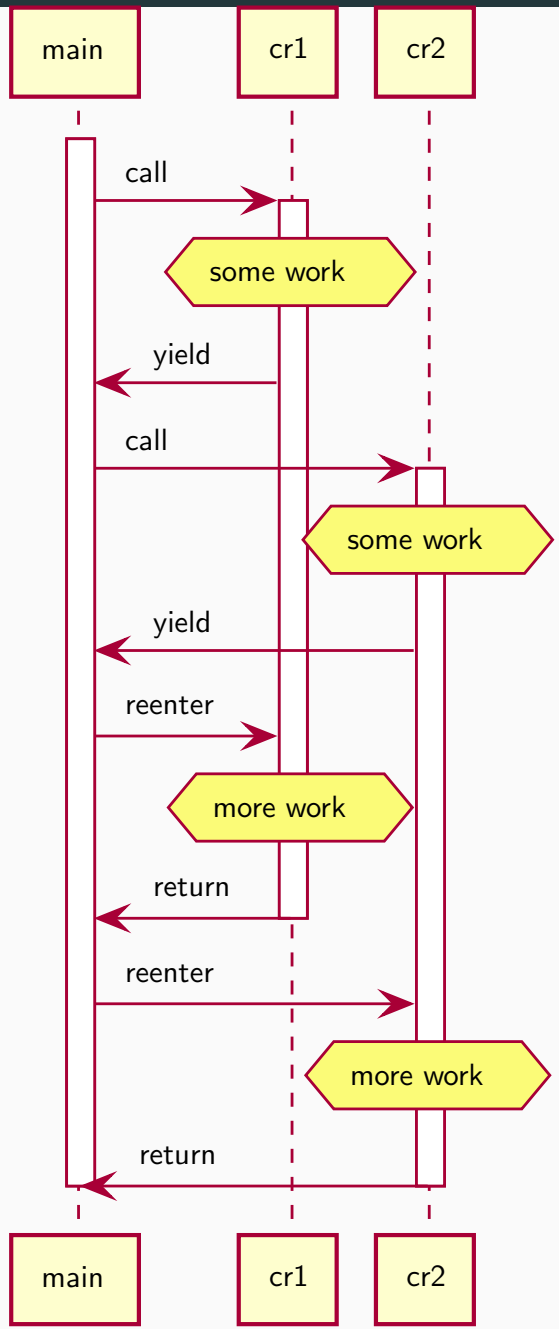
Coroutine

```
1  int main()
2  {
3    coroutine cr{&cr_impl};
4    cr.spawn();
5    //...main does something
6    cr.reenter();
7    return 0;
8  }
9
10 void cr_impl(yield_t &yield)
11 {
12   some_work();
13   yield();
14   more_work();
15 }
```



Coroutine concurrency

```
1  int main()
2  {
3     coroutine cr1{&cr_impl};
4     coroutine cr2{&cr_impl};
5     cr1.spawn();
6     cr2.spawn();
7     cr1.reenter();
8     cr2.reenter();
9     return 0;
10 }
11 void cr_impl(yield_t & yield)
12 {
13    some_work();
14    yield();
15    more_work();
16 }
```



Value yield example

```
1  int cr_impl(yield_t & yield)
2  {
3    yield(1);
4    yield(2);
5    return 4;
6  }
7
8  int main()
9  {
10   coroutine<int()> cr{&cr_impl};
11
12   assert(cr.spawn() == 1);
13   assert(cr.reenter() == 2);
14   assert(cr.reenter() == 4);
15   return cr.done() ? 0 : 1;
16 }
```

Argument passing example

```
1  int cr_impl(yield_t & yield, bool)
2  {
3    int value = 1;
4    while(yield(value))
5      value <<= 1;
6    return 0;
7  }
8
9  int main()
10 {
11   coroutine<int(bool)> cr{&cr_impl};
12
13   assert(cr.spawn(true) == 1);
14   assert(cr.reenter(true) == 2);
15   assert(cr.reenter(true) == 4);
16   return cr.reenter(false);
17 }
```

Properties

- Deterministic, no real concurrency

Properties

- Deterministic, no real concurrency
- Pass values out on yield in on reenter

Properties

- Deterministic, no real concurrency
- Pass values out on yield in on reenter
- May not be moved in memory

Properties

- Deterministic, no real concurrency
- Pass values out on yield in on reenter
- May not be moved in memory
- Can be stored (e.g. for deep sleep)

Properties

- Deterministic, no real concurrency
- Pass values out on yield in on reenter
- May not be moved in memory
- Can be stored (e.g. for deep sleep)
- Only constraint is the stack-size

Properties

- Deterministic, no real concurrency
- Pass values out on yield in on reenter
- May not be moved in memory
- Can be stored (e.g. for deep sleep)
- Only constraint is the stack-size
- Could be used as callback

Context switch

Main Context

```
1  int main()
2  {
3    coroutine cr;
4    cr.spawn()
5    /* make_context
6     *
7     *
8     *return from spawn*/;
9    cr.reenter()
10   /* switch_context
11    *
12    *return from reenter*/;
13   return 0;
14 }
```

Coroutine Context

```
1
2
3
4
5  void cr_impl()
6  {
7    yield()
8    /* switch_context
9     *
10    *return from yield*/;
11 }
12
13
14
```

Context switch

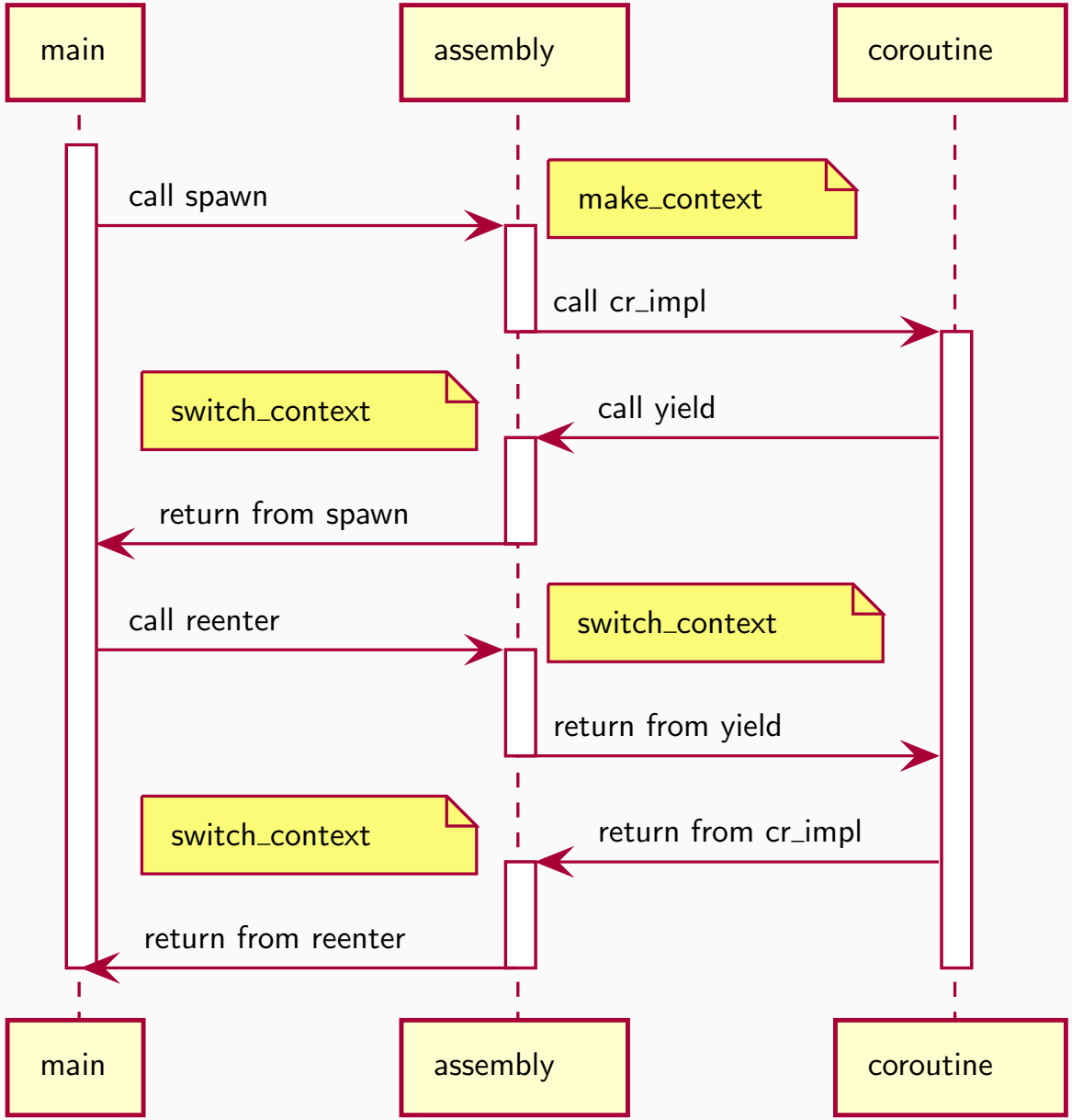
Main Context

```
1  int main()
2  {
3    coroutine cr;
4    cr.spawn()
5    /* make_context ──────────────────────→
6     *
7     *
8     *return from spawn*/; ←──────────────
9    cr.reenter()
10   /* switch_context ──────────────────→
11    *
12    *return from reenter*/; ←
13   return 0;
14 }
```

Coroutine Context

```
1
2
3
4
5  void cr_impl()
6  {
7    yield()
8    /* switch_context
9     *
10   *return from yield*/;
11 }
12
13
14
```

Context switch



Stack Pointer `SP`

- Points to lowest element on stack

Stack Pointer `SP`

- Points to lowest element on stack
- Decremented on function entry / incremented on exit (usually)

Stack Pointer `SP`

- Points to lowest element on stack
- Decremented on function entry / incremented on exit (usually)

Stack Pointer SP

- Points to lowest element on stack
- Decremented on function entry / incremented on exit (usually)

C++

```
1 void foo() { }
```

Stack Pointer SP

- Points to lowest element on stack
- Decremented on function entry / incremented on exit (usually)

C++

```
1 void foo() { }
```

ASM (by gcc)

```
1   str fp, [sp, #-4]!
2   add fp, sp, #0
3   sub sp, fp, #0
4   mov a1, a1 @ nop
5   sub sp, fp, #0
6   ldr fp, [sp], #4
7   bx lr
```

Link Register `LR`

- Points to the code location of function call

Link Register `LR`

- Points to the code location of function call
- Returning means jump to the location

Link Register `LR`

- Points to the code location of function call
- Returning means jump to the location
- Pushed on stack for new function call

Link Register LR

- Points to the code location of function call
- Returning means jump to the location
- Pushed on stack for new function call

C++

```
1 void foo() {bar();}
```

Link Register LR

- Points to the code location of function call
- Returning means jump to the location
- Pushed on stack for new function call

C++

```
1 void foo() {bar();}
```

ASM (by gcc)

```
1   push {fp, lr}
2   add fp, sp, #4
3   bl bar()
4   mov r0, r0 @ nop
5   sub sp, fp, #4
6   pop {fp, lr}
7   bx lr
```

Argument Register `a1-a4`

- Used to pass arguments in and return

Argument Register `a1-a4`

- Used to pass arguments in and return
- Large values store a reference

Argument Register `a1-a4`

- Used to pass arguments in and return
- Large values store a reference
- `a1` stores the return value

Argument Register `a1-a4`

- Used to pass arguments in and return
- Large values store a reference
- `a1` stores the return value
- Only valid in local context, not persistent

Argument Register `a1-a4`

- Used to pass arguments in and return
- Large values store a reference
- `a1` stores the return value
- Only valid in local context, not persistent
- `ip` is an additional scratch register

Argument Register `a1-a4`

- Used to pass arguments in and return
- Large values store a reference
- `a1` stores the return value
- Only valid in local context, not persistent
- `ip` is an additional scratch register

C++

```
1 int foo(int x, int y) { return x+y; }
```

Argument Register `a1-a4`

- Used to pass arguments in and return
- Large values store a reference
- `a1` stores the return value
- Only valid in local context, not persistent
- `ip` is an additional scratch register

C++

```
1 int foo(int x, int y) { return x+y; }
```

ASM (by gcc,-O3)

```
1   add a1, a1, a2
2   bx lr
```

Variable registers `v1-v8`

- Load/Store-Architecture

Variable registers `v1-v8`

- Load/Store-Architecture
- Values must be loaded into variable registers for operations

Variable registers `v1-v8`

- Load/Store-Architecture
- Values must be loaded into variable registers for operations
- Must be persistent after subroutine calls

Variable registers `v1-v8`

- Load/Store-Architecture
- Values must be loaded into variable registers for operations
- Must be persistent after subroutine calls
- When used, old values get pushed on the stack

Variable registers `v1-v8`

- Load/Store-Architecture
- Values must be loaded into variable registers for operations
- Must be persistent after subroutine calls
- When used, old values get pushed on the stack

C++

```
1  void bar(int&, int&,
2           int&, int&);
3  void foo(int &x, int &y,
4           int &z, int &i)
5  {
6      x = i+y+z;
7      bar(x,y,z,i);
8  }
```

Variable registers `v1-v8`

- Load/Store-Architecture
- Values must be loaded into variable registers for operations
- Must be persistent after subroutine calls
- When used, old values get pushed on the stack

C++

```
1  void bar(int&, int&,
2           int&, int&);
3  void foo(int &x, int &y,
4           int &z, int &i)
5  {
6      x = i+y+z;
7      bar(x,y,z,i);
8  }
```

ASM (by gcc)

```
1    push {v1, v2, v3, lr}
2    ldr v2, [a4]
3    ldr v1, [a2]
4    ldr lr, [a3]
5    mla ip, v1, v2, lr
6    str ip, [a1]
7    bl bar
8    pop {v1, v2, v3, lr}
9    bx lr
```

Context

- `r0- r3/a1-a4` Argument/Scratch register

Context

- `r0- r3/a1-a4` Argument/Scratch register
- `r4-r11/v1-v8` Variable registers

Context

- `r0- r3/a1-a4` Argument/Scratch register
- `r4-r11/v1-v8` Variable registers
- `r11/ fp` *Frame pointer on gcc*

Context

- `r0- r3/a1-a4` Argument/Scratch register
- `r4-r11/v1-v8` Variable registers
- `r11/ fp` *Frame pointer on gcc*
- `r12/ ip` Intra procedure scratch

Context

- `r0- r3/a1-a4` Argument/Scratch register
- `r4-r11/v1-v8` Variable registers
- `r11/ fp` *Frame pointer on gcc*
- `r12/ ip` Intra procedure scratch
- `r13/ SP` Stack pointer

Context

- `r0- r3/a1-a4` Argument/Scratch register
- `r4-r11/v1-v8` Variable registers
- `r11/ fp` *Frame pointer on gcc*
- `r12/ ip` Intra procedure scratch
- `r13/ SP` Stack pointer
- `r14/ LR` Link Register

Context

- `r0- r3/a1-a4` Argument/Scratch register
- `r4-r11/v1-v8` Variable registers
- `r11/ fp` *Frame pointer on gcc*
- `r12/ ip` Intra procedure scratch
- `r13/ SP` Stack pointer
- `r14/ LR` Link Register
- `r15/ PC` Program Counter

Creating a coroutine

```
1  struct coroutine
2  {
3    void * stack_pointer; //valid
4    template<typename Function>
5    Return spawn(Function && func)
6    {
7        return make_context(this, &func, &executor);
8    }
9    static void executor(coroutine * const this_,
10                         Function *func_p)
11   {
12     Function func = *func_p;
13     Return val = func({this_});
14     done = true;
15     switch_context(val, this_);
16   };
17 };
```

Make context

```
1  make_context_0:
2      @_make_context_0(cr* const, void *, void * )
3      @                  coroutine, func,   executor
4      @executor: (cr * const, void *)
5      @            coroutine,  func
6      push {v1-v8, lr} @push  the link register
7      mov v1, sp    @move the stack pointer to v1
8      ldr sp, [a1] @set the stack pointer
9      str v1, [a1] @store the old stack pointer
10
11     bx a3 @call the function
```

Context switch

```
1 void switch_context_0(void*& sp);
2 void yield  (void*& p){switch_context_0(p);}
3 void reenter(void*& p){switch_context_0(p);}
```

Context switch

```
1 void switch_context_0(void*& sp);
2 void yield  (void*& p){switch_context_0(p);}
3 void reenter(void*& p){switch_context_0(p);}
```

```
1 switch_context_0:
2   push {v1-v8, lr} @8
3
4   mov v1, sp    @1
5   ldr sp, [a1] @2
6   str v1, [a1] @2
7
8   pop {v1-v8, lr} @8
9   bx lr @1   --> overall 22
```

Context switch `uint32_t(uint64_t)`

```
1  uint64_t switch_context_1(uint32_t, void*& sp);
2  uint32_t switch_context_2(uint64_t, void*& sp);
3  uint64_t yield  (uint32_t i, void*& p)
4  {
5    return switch_context_1(i, sp);
6  }
7  uint32_t reenter(uint64_t i, void*& p)
8  {
9    return switch_context_2(i, sp);
10 }
```

Context switch `yield(uint32_t)`

```
1  switch_context_1:
2      @yield(uint32_t, void*) -> uint64_t
3      @       a1       , a2
4      push {v1-v8, lr}
5
6      mov v1, sp
7      ldr sp, [a2]
8      str v1, [a2]
9
10     pop {v1-v8, lr}
11
12     @reenter(uint64_t, void*) -> uint32_t
13     @return value is in a1
14     bx lr
```

Context switch `reenter(uint64_t)`

```
1  switch_context_2:
2      @reenter(uint64_t, void*) -> uint32_t
3      @         a1-a2    , a3
4      push {v1-v8, lr}
5
6      mov v1, sp
7      ldr sp, [a3]
8      str v1, [a3]
9
10     pop {v1-v8, lr}
11
12     @yield(uint32_t, void*) -> uint64_t
13     @return value is in a1-a2
14     bx lr
```

Statemachine example

```
1  struct statemachine
2  {
3    int state = 0;
4    bool done = false;
5    int operator()()
6    {
7      switch (state)
8      {
9        case 0:
10         state = 1; return f();
11       case 1:
12         state = 2; return g();
13       case 2:
14         done = true; return h();
15     }
16   }
17 };
```

Statemachine example

```
1  int coroutine(embo::yield_t<int()>& yield)
2  {
3      yield(f());
4      yield(g());
5      return h();
6  }
```

Statemachine Assembly

```
1  statemachine::operator()():
2    ldr a4, [a1]   @2
3    push {v1, lr}  @1
4    cmp a4, #0     @1
5    beq .L10       @1
6    cmp a4, #1
7    beq .L3
8    cmp a4, #2
9    beq .L4
10   pop {r4, lr}
11   bx lr
12 .L10:
13   mov a4, #1     @1
14   str a4, [a1]   @2
15   bl f()         @1
16   pop {v1, lr}   @1
17   bx lr          @1 -> 11 for f, 12 for g, 13 for h
```

Coroutine Assembly

```
1  coroutine(yield_t&): @14
2    push {v1, lr} @1
3    mov v1, a1    @1
4    bl f()        @1
5    mov a2, v1    @1
6    bl switch_context_1(int, void*) @1+22; 22 reentry
7    bl g() @1
8    mov a2, v1 @1
9    bl switch_context_1(int, void*) @1+22
10   bl h() @1
11   pop {v1, lr} @1
12   bx lr @1 -> 41 for f, 47 g, 47 for h
```

Statemachine stack example

```
1  struct statemachine {
2    int state = 0;
3    bool use_wchar = false;
4    array<char, 128> buffer;
5    array<wchar_t, 128> wbuffer;
6    void operator()() {
7      switch (state) {
8        case 0: state = 1; use_wchar = get_mode();
9        break;
10       case 1:
11         if (use_wchar) {state = 2; read(wbuffer);}
12         else           {state = 3; read( buffer);}
13         return;
14       case 2: handle(wbuffer); return;
15       case 3: handle( buffer); return;
16 } } };
```

Coroutine stack example

```
1  void cr(embo::yield_t<void()>& yield)
2  {
3    bool use_wchar = get_mode();
4    yield();
5    if (use_wchar)
6    {
7      array<wchar_t, 128> wbuffer;
8      read(wbuffer); yield();
9      handle(wbuffer);
10   }
11   else
12   {
13     array<char, 128> buffer;
14     read(buffer); yield();
15     handle(buffer);
16   }
17 }
```

Coroutine stack template example

```
1  template<typename Char>
2  void cr_impl(embo::yield_t<void()> &yield)
3  {
4    array<Char, 128> buffer;
5    read(buffer); yield();
6    handle(buffer);
7  }
8
9  void cr(embo::yield_t<void()>& yield)
10 {
11   bool use_wchar = get_mode();
12   yield();
13   if (use_wchar)
14     cr_impl<wchar_t>(yield);
15   else
16     cr_impl<char>(yield);
17 }
```

Summary

- www.github.com/klemens-morgenstern/embo.coroutine

Summary

- www.github.com/klemens-morgenstern/embo.coroutine
- klemens.d.morgenstern@gmail.com

Summary

- www.github.com/klemens-morgenstern/embo.coroutine
- klemens.d.morgenstern@gmail.com
- Any questions?