coroutinelab report

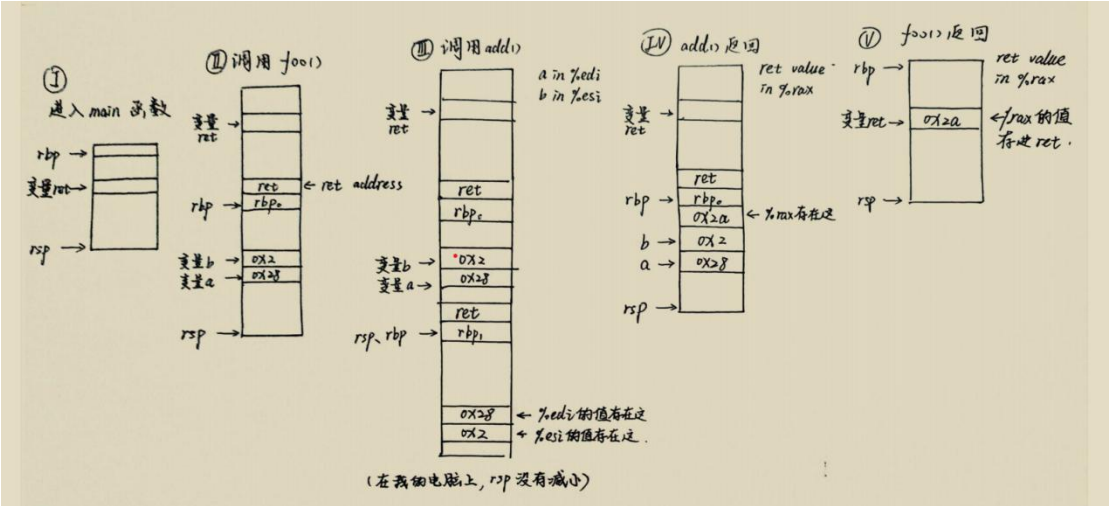
姓名：李增昊
学号：22307130108

问题回答：

part0，任务一：

函数调用过程：首先进入 main 函数，main 函数直接调用 foo()。foo()会调用函数 add()，并把两个临时变量 a，b 作为参数传入 add()。add()计算 a,b 的和，然后 add()结束，结果返回到 foo()。紧接着 foo()结束，结果返回 main，存在 main 函数的临时变量 ret 中，最后 main 调用标准输出函数 cout，将 ret 输出。

栈帧变化：



part0，任务二

1.一个普通函数支持 call 和 return 两个操作。

call 操作首先为下一步要运行的函数创建栈帧。然后，将正在进行的函数暂停，暂停时，要保存所有寄存器中的值。接着把返回地址写到新函数的栈帧中，将控制权交给新函数，从此新函数开始执行。

return 操作首先返回值存放到 caller 可以得到的地方。接着，它会销毁当前函数的调用栈，这个过程包括了销毁所有临时变量、参数、释放临时分配的内存。最终，它会把栈顶和栈底指针回复到 caller 栈帧的对应位置上，并从栈帧中读出返回地址，恢复 caller 的执行。

2.为什么我们说调用栈不能满足协程的要求？

使用调用栈时，只要一个函数返回了，它的调用栈就被销毁了，也就没有之后再“恢复”它的可能性。协程其实上就是要让一个函数可以在运行到一半时被挂起，等到需要时再恢复它，只使用调用栈的话，没办法保存恢复时必要的函数信息。

另外，调用栈一定只能返回到调用者。而对于无栈协程而言，不一定要返回给调用者。

3.协程支持三种基本操作：suspend,resume,destroy

suspend 操作将当前正在运行的函数挂起。并且会在 coroutine frame 里面保存必要的寄存器的值、临时变量，以及 suspend point 的具体位置，从而保证之后的 resume 或 destroy 可以正常进行。暂停后，可以将操作权转移给 caller，也可以直接恢复当前协程。如果操作权交给了 caller，那么该协程在栈上分配的空间就会释放，但是 coroutine frame 会被保存

resume 操作是对一个处于 suspend 状态的协程执行的。顾名思义，它会为即将恢复的协程重新分配栈帧（并把返回地址压栈），并从 coroutine frame 里读出 suspend point，从上次协程暂停的地方恢复协程的运行。当这个协程再次挂起或者结束时，resume 就返回了。

destroy 操作也只能对 suspended 的协程进行。它同样分配栈帧，但是却调用对应的析构函数，首先根据 suspend point 的信息，将局部变量全部释放，之后删除对应协程的 coroutine frame

4.对于每一个协程，可以为它初始化一个对应的结构体，在结构体中动态分配一块内存（动态分配的内存会放在 heap 上），在这块内存里保存好协程所需的信息。

part0，任务三

CPU 的状态包括了寄存器的值、栈顶和栈底指针、程序计数器 PC 的值（或者说 CPU 会改变它们的状态），以及 CPU 根据当前读到的指令，所确定的读写操作地址、ALU 执行怎样的操作等等。

在我看来，为了实现一个协程，比较简单粗暴的方法就是在一个协程挂起时，将它的栈帧保存起来，同时保存栈顶和栈底指针，PC，六个传参寄存器的值，所有 callee save 的寄存器值（不用保存 caller save 寄存器，因为当协程调用 suspend 时，自己会把这些寄存器保存到栈帧里）

part4，任务四

1.协程函数的返回值应该包含一个 promise 对象、对应于该协程的 coroutine handle、以及 operator co_await() + struct awaiter（也就是说这个返回值可以被 co_await）

2.promise 对象要提供以下函数：

```
get_return_object();
initial_suspend();
final_suspend();
unhandled_exception();
yield_value();
return_void();
```

有必要的还可能需要实现其他一些接口或重载。

3.awaitable object 应该实现以下接口：

```
operator co_await();
await_ready(); + await_suspend(); + await_resume();
```

4.coroutine handle 通常提供：

destroy();

resume();

promise();

done();

from_promise();（严格来说这不是 coroutine handle **对象**提供的）

等函数

5.co_return 和 co_yield 规定了 co_await 的协程挂起时的行为。用很短的语句代表了复杂功能的实现。co_await 可以不用关心它调用的操作具体是如何完成并返回给当前协程的。

6.

协程的调用过程：

以参考资料 understanding symmetric transfer 中的例子：

task foo(){ co_return;}

task bar(){co_await foo();}

为例。

（1）首先，bar 通过 co_await 调用了 foo。由于 foo 中使用了关键字 co_return，所以 foo 会被编译器当作协程。那么，程序要做的第一件事就是为 foo 分配 coroutine frame，复制参数（此处为空）到 coroutine frame 里，接着，就要构建 foo 的 promise 对象。

（2）在 foo 的 promise 对象里，调用 get_return_object();。这个函数会产生协程函数 foo 的返回值 task 对象。而 task 对象中就包含了 promise_type、coroutine handle、operator co_await、struct awaiter 等成员。

（3）foo 在 initial_suspend 处挂起。同时，task 对象被返回给 bar。

（4）bar 函数 co_await 返回值 task。这样就会调用 task 中的 operator co_await，产生 awaiter 对象，进而调用 await_ready,await_suspend,await_resume

（5）await_ready 用于判断是否要暂停当前协程 bar，切换到协程 foo。在本例中，await_ready 返回 false，表示 foo 函数需要被执行。

（6）然后 bar 被暂停，调用 await_suspend。协程 bar 的 coroutine handle 会作为参数传给 await_suspend，作为协程 foo 的 promise 对象中 std::coroutine_handle<promise_type> continuation 的值（这个成员表示了，foo 协程暂停后，下一步要恢复的协程）。之后，通过 foo 协程的 coroutine handle 上的.resume()方法，运行 foo。

（7）在协程 foo 中，依次经过 foo 函数的主体部分（如果有异常出现，还会进入异常处理部分）和 promise.final_suspend()。由于 foo 函数的主体只有一个 co_return，相当于调用了 promise.return_void()之后，直接跳转到 final_suspend()

（8）foo 协程在 final_suspend 处暂停，并通过调用 continuation.resume()，恢复协程 bar 的运行。

（9）在之后的过程中，可能会调用 foo 协程 task 对象的析构函数，析构过程是递归的，会先把 foo 的 coroutine frame,promise object,copies of arguments 析构，然后析构 task 对象

（10）最后调用 co_await 里的 await_resume()，产生 co_await 的返回值。在本例中，只是返回 void。之后，协程 bar 可谓是真正恢复运行，会开始执行 co_await 之后的语句。在本例中，没有更多语句要执行了，bar 会直接返回。

各个接口的简单功能解释：

（1）promise 对象中的接口：

get_return_object(): 产生协程函数返回值，也就是上例中的 task

initial_suspend(): 协程开始后的第一个暂停点，具体产生什么样的行为是自定义的。可以不挂起，直接继续协程的进行（如上例）；也可以挂起，等必要时 resume or destroy

unhandled_exception：处理协程函数主体中可能抛出的异常

final_suspend()：协程结束时的暂停点。基本任务是暂停当前协程，准备下一个协程的恢复

yield_value()：用于支持 co_yield val。暂停当前协程，并把 val 保存到 promise 对象里

return_void()\return_value()：用于支持 co_return 和 co_return val。return_void 基本什么也不用做。return_value 函数只要在 promise 对象中记录 val 就好。至于协程如何结束，可以交给 final_suspend 处理

（2）awaitable object 中的接口：

operator co_await()：可有，当然没有也可以。有的话，会根据这个接口创建 awaiter 对象，否则，awaitable 自己就当作是 awaiter。

await_ready()：判断是不是需要暂停当前协程，开始（恢复）另一协程。大多数情况下，都返回 false，表示需要。但如果另一协程已经执行完了，就返回 true。

await_suspend()：根据其返回值类型，有多个版本，也就造成了 symmetric transfer 和 non symmetric transfer 的差异。但是简单来说，就是要暂停当前协程，保存好当前协程的 coroutine handle，以便后续的恢复，然后 resume 另一协程。

await_resume()：产生 co_await 的返回值，一般放回一个 void 就行了

（3）coroutine handle 中的接口：

P.S.我还是只写了实验中用到过的那几个接口，其他的由于不是很熟悉+时间关系，留到假期再好好了解一番。

resume()：恢复协程运行

destroy()：销毁协程 coroutine handle 和相关状态

done()：用来表示一个协程是否已经结束（suspended at final suspend point）

promise()：获取 coroutine handle 所属协程的 promise 对象

from_promise()：根据 promise 对象获得 coroutine handle 实例

各部分通过的截图

```
simpson@simpsons:/mnt/c/Users/李增昊/Desktop/computer system$ cd libco\ -\ TODO/
simpson@simpsons:/mnt/c/Users/李增昊/Desktop/computer system/libco - TODO$ cd libco_v1
simpson@simpsons:/mnt/c/Users/李增昊/Desktop/computer system/libco - TODO/libco_v1$ ./main
test-1 passed
test-2 passed
test-3 passed
Congratulations! You have passed all the tests of libco-v1!
simpson@simpsons:/mnt/c/Users/李增昊/Desktop/computer system/libco - TODO/libco_v1$ cd ..
```

```
simpson@simpsons:/mnt/c/Users/李增昊/Desktop/computer system/libco - TODO$ cd libco_v2
simpson@simpsons:/mnt/c/Users/李增昊/Desktop/computer system/libco - TODO/libco_v2$ ./main
test-1 passed
test-2 passed
test-3 passed
test-4 passed
Congratulations! You have passed all the tests of libco-v2!
simpson@simpsons:/mnt/c/Users/李增昊/Desktop/computer system/libco - TODO/libco_v2$ cd ..
```

```
simpson@simpsons:/mnt/c/Users/李增昊/Desktop/computer system/libco - TODO$ cd libco_v3
simpson@simpsons:/mnt/c/Users/李增昊/Desktop/computer system/libco - TODO/libco_v3$ ./main
libco_v3 test passed!
simpson@simpsons:/mnt/c/Users/李增昊/Desktop/computer system/libco - TODO/libco_v3$ cd ..
```

```
simpson@simpsons:/mnt/c/Users/李增昊/Desktop/computer system/libco - TODO/libco_v4$ ./main
Hello, ICS 2023!
2 3 5 8 13 21 34 55 89 144
```

```
simpson@simpsons:/mnt/c/Users/李增昊/Desktop/computer system/libco - TODO$ cd libco_v5
simpson@simpsons:/mnt/c/Users/李增昊/Desktop/computer system/libco - TODO/libco_v5$ ./main
Start libco_v5 test
libco_v5 task test passed!
simpson@simpsons:/mnt/c/Users/李增昊/Desktop/computer system/libco - TODO/libco_v5$
```

libco 代码和简要的实现思路

libco_v1:

整体思路：借助 ucontext.h 中的工具，利用 swapcontext 实现协程的切换。协程的栈帧在 coroutine 结构体中进行保存。在我所需要编写的函数中，只需要根据协程的调用栈获取协程的调用信息，做出合理的切换即可。为此，我需要维护一个协程调用栈。特别的，在协程第一次被调用时，还需要为它分配合适的上下文。（比如栈帧、协程结束后运行哪一协程）

coroutine 结构体：

```
struct coroutine {
    bool started = false;
    bool end = false;

    func_t coro_func = nullptr;
    void* args = nullptr;

    // TODO: add member variables you need
    char stack[1024*1024];
    ucontext_t ctx = {0};
    int state=0;//-1 indecates the end of coroutine

    coroutine(func_t func, void* args) : coro_func(func), args(args) {
        /* TODO */
        //actually don't need to write anything
    }

    ~coroutine() {
        /* TODO */
    }
};
```

结构体中的 stack 用于保存协程的栈帧

state 用于记录 resume,yield 的参数，以便产生正确的返回值。

对于构造函数和析构函数，其实可以什么也不做。（我选择在 resume 函数里才为 coroutine 结构体中的 ctx 变量初始化）

coroutine_env 类：

```
class coroutine_env {
private:
    // TODO: add member variables you need
    coroutine* stack[10]={nullptr}; //the stack stores coroutines
    int top=-1;
public:
    coroutine_env() {
        // TODO: implement your code here
        coroutine* main_coro=create(nullptr,nullptr);
        push(main_coro);
    }
    coroutine* get_coro(int idx=-1) {
        // TODO: implement your code here
        if(idx>=0&&idx<=top)
            return stack[idx];
        else
            return stack[top];
    }
    void push(coroutine* co) {
        // TODO: implement your code here
        assert(top<9);
        stack[++top]=co;
    }
    void pop() {
        // TODO: implement your code here
        stack[top]=nullptr;
        top--;
    }
};
```

内部使用了一个 coroutine*数组（栈）用来记录协程的调用信息

在构造函数中，会将 main 函数的协程压进栈。在这里调用了 create 函数，实际上只是对 coroutine 结构体中的函数指针、函数参数指针赋值而已。对于 main 函数而言，可以让它们为 nullptr，因为 main 函数的调用不会利用这两个成员变量实现

pop 和 push 操作的实现比较简单，这里不多讲

get_coro 操作支持参数的默认初始化，在调用者没用给出参数时，相当于 get_top 操作。

create 和 release 函数如下，简单调用 new 和 delete 就可以

```
coroutine* create(func_t func, void* args) {
    // TODO: implement your code here
    coroutine* co=new coroutine(func,args);
    return co;
}

void release(coroutine* co) {
    // TODO: implement your code here
    delete co;
}
```

resume 函数：

```
int resume(coroutine* co, int param) {
    // TODO: implement your code here
    if(co->state==-1)//ended
        return -1;

    coroutine* caller=g_coro_env.get_coro();
    if(!co->started){
        getcontext(&co->ctx);//initialize
        co->ctx.uc_link=&(caller->ctx);
        co->ctx.uc_stack.ss_sp=co->stack;
        co->ctx.uc_stack.ss_size=1024*1024;
        co->ctx.uc_stack.ss_flags=0;
        makecontext(&co->ctx,(void(*)())func_wrap,1,co);

        co->started=true;
    }
    co->state=param;
    g_coro_env.push(co);
    swapcontext(&caller->ctx,&co->ctx);

    return co->state;
}
```

在协程第一次 resume 时，协程才会被调用。这时会更改 started 标签，并为该协程分配上下文：

先通过 getcontext 为协程的 ucontext_t 成员初始化。然后 uc_link 指向协程调用者的 ucontext_t 结构体，表示该协程挂起后，会返回到调用者处。栈指针当然要指向协程结构体中已经分配好的栈空间。最后，使用 makecontext，使得程序切换到该上下文的时候，会跳转到 func_wrap 进行

之后要做的就是将该协程压进栈，并且调用 swapcontext 实现协程的切换

在这里注意一下 state 的更新问题。resume 的参数会直接更新 state，以便 yield 函数获得正确的返回值，这个返回值就是上一个 resume 传入的参数。同时，yield 也会根据自己接受到的参数更新 state，当 resume 返回时，其返回值应该是 yield 更新过后的 state

yield 函数：

```
int yield(int ret) {
    // TODO: implement your code here
    coroutine* cur=g_coro_env.get_coro();
    g_coro_env.pop();
    coroutine* caller=g_coro_env.get_coro();

    int temp=cur->state;
    cur->state=ret;
    swapcontext(&cur->ctx,&caller->ctx);

    return temp;
}

}  // namespace coro
```

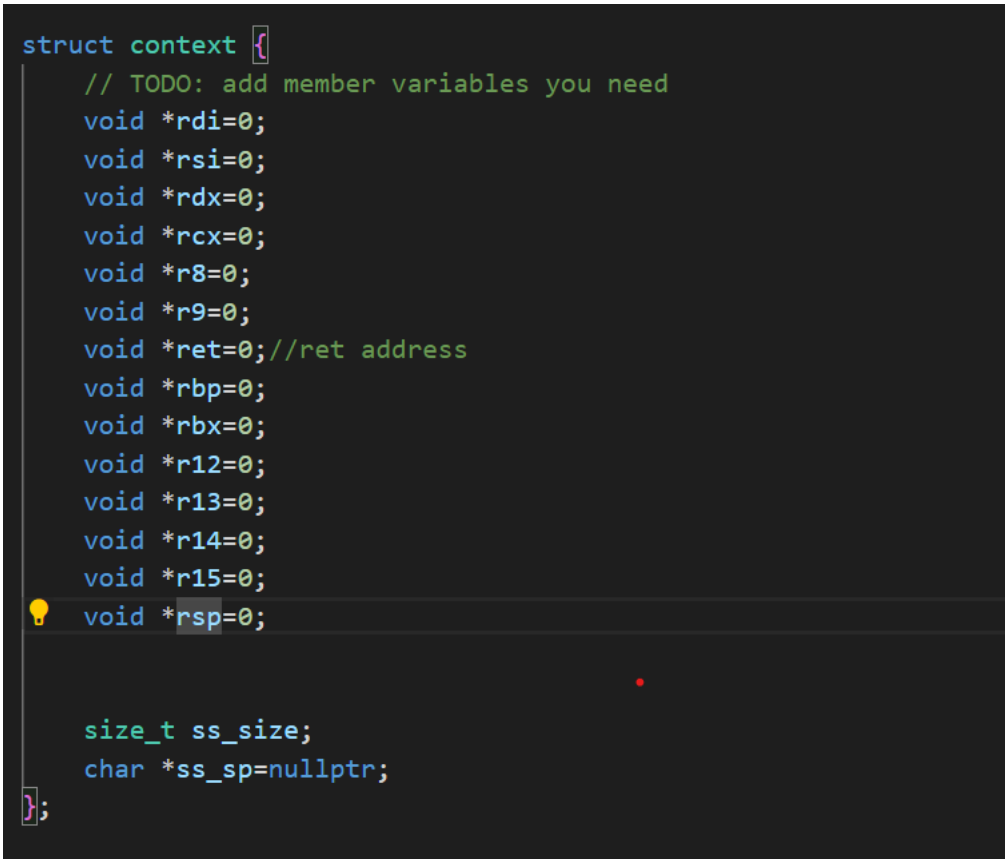
通过协程调用栈获取当前协程的调用者。之后要做的事情和上面讲述的一样：更新 state，并且切换协程。

libco_v2:

首先是 ucontext 结构体的实现。里面应该包含一个栈指针，用来指向该协程所使用的栈空间。既然有了栈指针，自然应该有一个参数记录栈的大小。

剩下的，我们需要一些成员变量用来记住协程切换时寄存器的值。在这里，我们只要记住 6 个传参寄存器和 callee save 寄存器即可。对于 caller save 寄存器，协程在调用 coro_ctx_swap 时会自己保存好，我们不需要专门记录。

用于保存寄存器值的成员变量类型为 void*，其实任何有 64 位的变量都可以



之后是 coro_ctx_swap。这个函数会保存一个协程的上下文到 context 结构体，并将程序切换到另一个协程。先来看保存上下文的部分：

第一个参数是我用于保存上下文的 context 结构体指针，也就是 rdi

这一段代码就是利用 pushq 操作，将寄存器的值存放到 context 结构体中。我们重点看一下 rip（PC）是如何保存的。

假设之后要恢复该上下文，程序应该从哪里开始执行呢？答案显然是协程发生切换的位置。所以 PC 应该保存为 coro_ctx_swap 的返回地址处。这样，要恢复的协程就会从上一次切换的位置开始，直接执行 coro_ctx_swap 结束后的第一条指令。coro_ctx_swap 的返回地址会在%rsp 中，这是因为 call 操作会将返回地址压进栈（并将 rsp 减 8）。而 rsp+0x8 就是上一函数的栈底指针。

切换上下文的部分就更好写了，我只需要利用 popq 操作，顺序将 context 结构体中的变量值赋给对应寄存器即可。注意 rip 不能直接赋值，而是先将 ret 赋给 rax，最后再无条件跳转到 rax 所示地址处

```
coro_ctx_swap:
leaq 8(%rsp),%rax
leaq 112(%rdi),%rsp
pushq %rax
pushq %r15
pushq %r14
pushq %r13
pushq %r12
pushq %rbx
pushq %rbp
pushq -8(%rax)
pushq %r9
pushq %r8
pushq %rcx
pushq %rdx
pushq %rsi
pushq %rdi
```

```
movq %rsi, %rsp
popq %rdi
popq %rsi
popq %rdx
popq %rcx
popq %r8
popq %r9
popq %rax
popq %rbp
popq %rbx
popq %r12
popq %r13
popq %r14
popq %r15
popq %rsp
jmp *%rax
```

最后可以写出 ctx_make 的代码。其中对 rsp(结构体中的)进行了向“8”的对齐操作。ctx_make 的作用等于 makecontext，切换到此上下文时会跳转到指定的函数处开始执行。

```
namespace coro {

void ctx_make(context* ctx, func_t coro_func, const void* arg) {
    // TODO: implement your code here
    char *sp=ctx->ss_sp+ctx->ss_size;
    sp=(char*)((unsigned long)sp&-16LL);
    ctx->rsp=sp-8;

    ctx->ret=(char *)coro_func;
    ctx->rdi=(char *)(arg);
}

}  // namespace coro
```

以上大致实现了 v1 中 swapcontext 和 makecontext 的功能。

libco_v2:共享栈的实现

写在最前面：对于 share_stack 中的 count，在我的实现中，它指明了 stack_mem 数组中元素的个数。（我**暂且**认为我对 count 的含义没有理解错）而且，stack_mem 数组中元素的个数在一个 share_stack 结构体被创建之后，**就不再改变**，除析构函数外的所有函数都不会主动增加（减少）share_stack 中 stack_mem 的个数。

这么做的原因是，我很难在某一个函数体中判断此时增加（或减少）数组中的 stack_mem **是否真的**会提高程序的性能（比如减少 memcpy 的次数、减少空间的使用等等）。因为增加 stack_mem 有可能会减少 memcpy 的次数，但是一定会增加空间的使用；相反，减少 stack_mem 可能会减少空间的使用，但是也增加了不同协程使用 share_stack 中同一个 stack_mem 的概率，也就意味着协程恢复和挂起时 memcpy 的次数可能会增加。这本身是一个权衡的过程。权衡的结果不仅仅取决于当前有多少个协程，也跟各个协程的使用频率、协程的时序关系有关，因此相当难判断。

所以，干脆让协程库的使用者决定到底要为 share_stack 分配多少 stack_mem。使用者可以大致预估他将要使用的协程个数，协程的使用频率和时序关系，选择一个比较合适的个数。（当然也可以增加一个接口，让使用者可以动态地增加 share_stack 中 stack_mem 的个数，或者删除一些没有协程正在使用的 stack_mem）

但是，对 count 的含义还可能有其他解释，上面的选择也未必就非常合理。如果助教大哥发现我对 count 的理解有很大误区，或是对上述选择有疑问的话，希望可以和我联系一下再考虑是否扣分~（毕竟协程库的实现本身也是一个创作和思考的过程，可能有不同的想法）。

如果看完了上面一大段烦人的文字之后，认为我的理解没有太大问题，那么接下来就可以看看我的具体实现：

stack_mem 结构体：

```
struct stack_mem {
    int stack_size = 0;
    // TODO: add member variables you need
    char *sp=nullptr;

    coroutine* last_user=nullptr;

    stack_mem(size_t size) : stack_size(size) {
        // TODO: implement your code here
        sp=new char[size]();
    }

    ~stack_mem() {
        // TODO: implement your code here
        last_user=nullptr;
        if(sp!=nullptr)
            delete sp;
        sp=nullptr;
    }
};
```

构造和析构函数无需多言。重点看看 last_user 成员，它是一个指针，指向该 stack_mem 的上一个使用者，用于减少不必要的 memcpy，具体作用在函数 swap 中会看到。

share_stack 结构体：

```
struct share_stack {
    // TODO: add member variables you need
    int count = 0;
    int idx=-1;
    int stack_size = 0;
    stack_mem **stack_array = nullptr;

    share_stack(int count, size_t stack_size)
        : count(count), stack_size(stack_size) {
        // TODO: implement your code here
        assert(count>=1);
        stack_array=new stack_mem*[count];
        for(int i=0;i<count;i++)
            stack_array[i]=new stack_mem(stack_size);
    }
    ~share_stack() {
        // TODO: implement your code here
        for(int i=0;i<count;i++)
            delete stack_array[i];
        delete[] stack_array;
    }
    stack_mem *get_stackmem() {
        // TODO: implement your code here
        assert(stack_array!=nullptr&&count!=0);
        idx=(idx+1)%count;
        return stack_array[idx];
    }
};
```

构造函数就是创建一个 stack_mem 数组，数组中每个元素的大小就是 stack_size 的大小。

get_stackmem 函数用于从 stack_mem 数组中选择一个元素，分配给某一协程使用。这里**比较粗糙地**轮次调用数组中的每一个元素。其实可以为函数增加一个带默认初始化的参数，让使用者有机会选择 stack_mem。但是，由于 get_stackmem 只会在 create 中调用，而 create 也没有额外参数，即使有，也会一定程度上破坏封装性，所以就采用了以上比较粗糙的策略（**正是因为在这里我无法找到一个满意的策略，才使我怀疑我对这部分的实现可能有误解，所以才有了“写在最前面”那一大段话**）

coroutine 结构体：

```
struct coroutine {
    bool started = false;
    bool end = false;

    stack_mem* stk_used=nullptr;

    func_t coro_func = nullptr;
    void *arg = nullptr;

    // TODO: add member variables you need
    stack_mem *stk=nullptr;
    context ctx = {0};
    int data=0;
    long long size_has_used=0;

    ~coroutine() {
        // TODO: implement your code here
        if(stk_used!=nullptr){
            stk_used->last_user=nullptr;
            stk_used=nullptr;
        }
        if(stk!=nullptr&&stk->sp!=nullptr)
            delete stk;
    }
};
```

stk_used：如果该协程使用了共享栈，那么 stk_used 就会指向该协程使用的 stack_mem 数组元素。如果没使用共享栈，该变量为 nullptr。它与

stk：它会指向协程使用的私有栈（若有）。如果是使用共享栈的协程，那么协程挂起时，stk 指向的空间会是用来保存该协程栈信息的空间。

size_has_used：用来记住当前协程的栈空间大小。以便分配一块刚好大小的空间。

coroutine_env 结构体：

与 v1 中基本一致，这里不再展示代码了。

create 函数：

与 v1 中差别较大的是，我需要在 create 函数中判断协程是否使用了共享栈，并且简单的判断一下 stack_size 是否合理。

下面是前半部分，处理了使用私有栈的状况。

```
coroutine* create(func_t coro_func, void* arg, const coroutine_attr* attr) {
    coroutine_attr at;
    if (attr != nullptr) {
        at = *attr;
    }
    // TODO: implement your code here
    coroutine* co=new coroutine;
    co->ctx={0};
    co->arg=arg;
    co->coro_func=coro_func;

    if(at.sstack==nullptr){//not using share-stack
        if(at.stack_size>(1<<17))
            at.stack_size=1<<17;
        if(at.stack_size<(1<<13))
            at.stack_size=1<<13;
        at.stack_size=at.stack_size&(~(1<<12)+1);
        co->stk=new stack_mem(at.stack_size);
        co->stk_used=nullptr;
        co->ctx.ss_sp=co->stk->sp;
        co->ctx.ss_size=co->stk->stack_size;
    }
    else{//using share_stack
```

如果 at.sstack 为 nullptr，就表明使用的是私有栈。程序会对栈空间的大小做边界限定，然后申请对应的空间。之后让 co->stk 指向私有栈。同时为 co->ctx 的栈指针、栈大小变量赋值。

然后是处理共享栈的部分：

```
    }
    else{//using share_stack
        at.stack_size=at.sstack->stack_size;
        if(at.stack_size>(1<<17)){
            int count=at.sstack->count;
            delete at.sstack;
            at.sstack=new share_stack(count,1<<17);
        }
        if(at.stack_size<(1<<13)){
            int count=at.sstack->count;
            delete at.sstack;
            at.sstack=new share_stack(count,1<<13);
        }

        co->stk=nullptr;
        co->stk_used=at.sstack->get_stackmem();
        co->ctx.ss_sp=co->stk_used->sp;
        co->ctx.ss_size=co->stk_used->stack_size;

    }
    return co;
}
```

co->stk 为 nullptr，而 co->stk_used 指向使用的共享栈。

swap 函数：

```
void swap(coroutine* curr, coroutine* pending) {
    // TODO: implement your code here
    if(curr->stk_used!=nullptr){
        int get_stk_size=0xff;
        long long size=(curr->ctx.ss_sp)+(curr->ctx.ss_size)-(char*)(&get_stk_size);
        size=(size+0x1<<4)&(-16LL);
        curr->size_has_used=size;
    }
    if(pending->stk_used!=nullptr&&pending->stk_used->last_user!=pending){
        save_stack(pending->stk_used->last_user);
        pending->stk_used->last_user=pending;
        if(pending->stk!=nullptr){
            char* dest=(pending->ctx.ss_sp)+(pending->ctx.ss_size)-(pending->size_has_used);
            char* src=pending->stk->sp;
            assert(pending->stk->stack_size>0);
            memcpy(dest,src,pending->stk->stack_size);
        }
    }
    coro_ctx_swap(&curr->ctx,&pending->ctx);
}
```

如何知晓当前协程用了多少栈空间：

这一步是通过定义一个临时变量，从而获知当前栈底的大概位置的。虽然会有一点偏差，但一定会把需要的部分的大小包括在内。这个大小会被 size_has_used 记住。栈空间暂且不做保存。

什么时候保存栈空间：

只有当下一个共享栈使用者要用到 share_stack 中同一个 stack_mem 时才会保存。这一步是通过上面的第二个 if 语句实现的。同步要做的：修改 last_user，以及把下一使用者保存的栈空间 memcpy 回到共享栈中。

最后就是调用 coro_ctx_swap。

save_stack 函数：

```
void save_stack(coroutine* co) {
    // TODO: implement your code here
    if(co==nullptr) return;
    if(co->end) return;
    assert(co->size_has_used!=0);
    if(co->stk==nullptr)
        co->stk=new stack_mem(co->size_has_used);
    assert(co->stk!=nullptr);
    char* src=(co->ctx.ss_sp)+(co->ctx.ss_size)-(co->size_has_used);
    assert(co->size_has_used>0);
    memcpy(co->stk->sp,src,co->size_has_used);
    co->stk->stack_size=co->size_has_used;
}
```

在没有上一个使用者，或者 co 已经结束时，不会执行。

在其他情况下，会 new 一块刚好大小的空间，用来保存协程挂起时的栈空间

resume，yield，release 函数与 v1 中相差不大，不过多解释。

libco_v3：

这一部分直接参考 coroutinelab 文档中给出的示例

四个宏的作用分别是

CO_BEGIN：标记协程已经开始，并且用 switch 语句判断跳转点

CO_YIELD(a)：让协程挂起，并返回参数 a。协程恢复时会跳转到此处

CO_RETRUN(a)：返回参数 a，并标记协程已经结束。下一次协程恢复时，会直接跳转到 CO_END

CO_END：与 CO_BEGIN 对应，标记协程结束（因为可能有不写 CO_RETURN）

下面是代码：

```
#define CO_BEGIN          \
    if(!begin)            \
    {begin=true;start=0;}\
    switch(start){        \
    case 0:


#define CO_END            \
        case -1:          \
            end=true;     \
    }

#define CO_YIELD(a) start=__LINE__;return a;case __LINE__:;

#define CO_RETURN(a) end=true; start=-1; return a;
```

```
struct coroutine_base {
    /* TODO */
    int start=0;
    bool begin=false;
    bool end=false;
};

class fib : public coroutine_base {
private:
    /* TODO */
    int ret=0;
    int b=1;
public:
    // TODO: update below code when you implement
    // CO_BEGIN/CO_END/CO_YIELD/CO_RETURN
    int operator()() {
        CO_BEGIN
            while (1){
                CO_YIELD(ret)
                int temp=ret;
                ret=b;
                b=temp+b;
            }
        CO_END
        return -1;
    }
};
```

libco_v4：

主体部分就是实现一个 generator。

最基本的，generator 应该要含有对应协程的 coroutine handle，提供 resume、done 等函数。

```
private:
    // TODO: implement generator constructor
    explicit generator(handle coro_in) noexcept {coro=coro_in;}

    // TODO: add member variables you need
    handle coro;
};
```

按照 *Understanding the promise type* 中给出的解释，我的 generator 应该包含一个 promise_type，以支持协程的 promise 接口。并且在 promise_type 中，需要制定协程本身的行为方式，比如调用、返回和发生异常时的行为。

为了支持 co_yield，promise_type 中还应该有 yield_value 函数

```
// TODO: implement promise_type
struct promise_type;
using handle = std::coroutine_handle<promise_type>;
struct promise_type{
    public:
        Value data;

        promise_type():data(0){}

        generator<Ref,Value> get_return_object(){return generator(this->co());}
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void unhandled_exception() { std::terminate(); }
        void return_void(){}
        std::suspend_always yield_value(Value value) noexcept{
            data = value;
            return {};
        }
        auto co() { return handle::from_promise(*this); }
        Value& getvalue(){return data;}
};
```

get_return_object：promise_type 必须要有这个成员函数，用来给协程的调用者提供对应的接口、句柄，例如 coroutine handle。这些接口或句柄都定义在 generator 对象中，因此，返回一个带对应协程句柄的 generator 对象即可。

initial_suspend\final_suspend：简单设计为协程被调用和返回时都会挂起。回到调用者处。

unhandled_exception：调用 std::terminate()，直接终止。

yield_value：首先需要保存参数的值，使得 resume 时可以得到对应的数据。在这里，**比较粗暴地**使用赋值的方法，传递给一个成员变量。**在 v5 中会对此进行优化**。

然后，应该暂停协程。

getvalue：用于在 resume 时，获取 yield 接收的数据。

协程的恢复是通过 iterator 进行的。iterator++就是 resume。当 resume 返回时，就会用 operator*获取 co_yield 接收的数据。operator==只会用在判断迭代器是否等于 generator.end()，以支持 auto &a:b。其功能就是判断一个协程是否已经完成，有没有执行的必要；或者协程句柄是否存在。所以只要形式上支持与另一个对象比较就可以了，empty 就是这样一个空类，支持比较的形式，而 operator 的返回值就是!coro_||coro_.done()

```
    ~iterator() {}

    // TODO: implement operator== and operator!=
    bool operator==(const empty&) const noexcept{
        return !coro_||coro_.done();
    }
    bool operator!=(const empty&) const noexcept{
        return coro_&&!coro_.done();
    }
    // TODO: implement operator++ and operator++(int)
    iterator& operator++(){
        coro_.resume();
        return *this;
    }
    void operator++(int){
        coro_.resume();
    }
    // TODO: implement operator* and operator->
    Value& operator*(){
        return coro_.promise().getvalue();
    }
    Value* operator->(){
        return &(coro_.promise().getvalue());
    }
```

```
private:
    friend generator;

    // TODO: implement iterator constructor
    // hint: maybe you need to a promise handle
    explicit iterator(handle p) noexcept {coro_=p;}

    // TODO: add member variables you need
    handle coro_;
};
```

有了这些比较容易写出 begin 和 end，以支持 iterator 的用法

```
// TODO: implement begin() and end() member functions
iterator begin(){
    auto iter=iterator(coro);
    ++iter;
    return iter;
}

empty end(){
    return{};
}
```

libco_v5：

实现可递归的 generator：

最主要就是支持 co_yield 一个同类型的 generator。参考普通函数的调用过程，可递归的 generator 也应该有一个类似于栈（或者调用树）的结构。通过最内层的协程可以知道上一层协程是什么，并一直这样递归到最外层的协程。为了加快协程恢复时的速度，最外层的协程也应该要能够获知最内层的协程是什么，以便用 O(1)的速度恢复该协程。

```
struct promise_type;
using handle=std::coroutine_handle<promise_type>;
struct promise_type{
    friend generator;
    public:
        union{
            promise_type* leaf_;
            promise_type* root_;
        };
        std::add_pointer_t<Ref> data;
        promise_type* parent_;


        promise_type():root_(this){}
```

为支持 co_yield 一个 generator，应该增加一个 yield_value(generator&& g)的接口。同时，这个接口返回的 awaitable 对象不再是简单的 std::suspend_always，我需要定制对应的 awaiter，管理协程调树的信息。

再 awaiter 对象中，await_ready 返回！g_.coro。表示如果 co_yield 的协程已经结束，就可以忽略本次 co_yield。

await_suspend 中对调用树进行调整。最后返回下一协程的句柄，程序很快会 resume 它。

```
    //co_yield a generator
    struct seq_awaitar{
        generator g_;

        explicit seq_awaitar(generator&& g):g_(std::move(g)){}
        bool await_ready() noexcept {return !g_.coro;}
        std::coroutine_handle<> await_suspend(handle h) noexcept {
            auto& cur=h.promise();
            auto& nested=g_.coro.promise();
            auto& root=cur.root_;

            nested.root_=root;
            nested.parent_=&cur;
            root->leaf_=&nested;

            return g_.coro;
        }
        void await_resume() noexcept {}
    };
    seq_awaitar yield_value(generator&& g) noexcept {
        return seq_awaitar{std::move(g)};
    }
```

当一个协程运行结束，调用 final_suspend 的时候，我也要对协程调用树做出调整。具体表现为，调整 leaf（最内层协程）为当前协程的 parent（上一协程）。如果上一协程不存在，就意味着这一部分的递归调用已经结束，不需要再做什么了，返回 std::noop_coroutine()，等待下一步指令。

```
//final suspend
struct final_awaitar{
    bool await_ready() noexcept {return false;}
    std::coroutine_handle<> await_suspend(handle h)noexcept{
        auto& promise=h.promise();
        auto parent=h.promise().parent_;
        if(parent){
            promise.root_->leaf_=parent;
            return handle::from_promise(*parent);
        }
        else return std::noop_coroutine();
    }
    void await_resume() noexcept {}
};
final_awaitar final_suspend() noexcept {
    return final_awaitar{};
}
```

最后是 resume，我们希望 resume 的是最内层协程（leaf）

```
void resume(){handle::from_promise(*leaf_).resume();}
```

P.S.对普通 co_yield 的改动（yield_value(Ref&& value)）：

还记得再 V4 中，是直接赋值给 promise_type 内部成员变量，从而记住 co_yield value 中 value 的值。这意味着其中有一次拷贝或者移动，在阅读了一些参考资料后，我意识到这是可以避免的。由于 co_yield 的临时变量会被保存在 coroutine frame 中，可以用指针来避免这一次拷贝。

```
std::suspend_always yield_value(Ref&& value) noexcept {
    root_->data=std::addressof(value);
    return {};
}
```

相应的，data 的声明改为 std::add_pointer_t<Ref> data

之后迭代器部分也稍作改动。iterator++要调用的是 promise_type 中自定义的 resume。operator*，operator->也会随着 co_yield value 的改动而修改。

```
// TODO: implement operator== and operator!=
friend bool operator==(const iterator& it,empty) noexcept {
    return (!it.coro_)||it.coro_.done();
}
friend bool operator!=(const iterator& it,empty) noexcept{
    return it.coro_&&(!it.coro_.done());
}
// TODO: implement operator++ and operator++(int)
iterator& operator++(){
    coro_.promise().resume();
    return *this;
}
void operator++(int){
    (void)operator++();
}
// TODO: implement operator* and operator->
reference operator*()const noexcept{
    return static_cast<reference>(*coro_.promise().data);
}
pointer operator->()const noexcept
requires std::is_reference_v<reference>{
    return std::addressof(operator*());
}
```

sleep.h 的实现：

根据我的理解，执行 co_await sleep{}的时候，只需要让程序暂停一定的时长。在暂停结束之后，并不会立即恢复之前的协程，而是把该协程 push 到 task_queue 中。当程序运行到 main 函数末端，执行 coro::wait_task_queue_empty()的时候，才会恢复处于队头的协程。

首先，在 Task 中完成 promise_type。因为 task 会使用 co_await，所以会被编译器当作协程编译，promise_type 就必不可少。

```
struct Task {
    // TODO: add functions to make Task be an coroutine handle
public:
    struct promise_type;
    using handle=std::coroutine_handle<promise_type>;
    struct promise_type{
        Task get_return_object()noexcept{
            return Task{};
        }
        void return_void()noexcept{}
        std::suspend_never initial_suspend()noexcept{return {};}
        void unhandled_exception(){std::terminate();}
        std::suspend_always final_suspend() noexcept {return {};}
    };
};
```

由于 sleep{} 是一个 awaitable 对象，所以其中至少要实现 await_ready,await_suspend,await_resume 三个函数。我需要自定义 await_suspend，让它可以实现“暂停特点时长，并把协程入队”的功能。

```
    // TODO: add functions to make sleep be an awaitable object

    bool await_ready()noexcept{return false;}
    void await_suspend(std::coroutine_handle<> h) noexcept {
        std::this_thread::sleep_for(delay);
        my_queue.push(h);
    };
    void await_resume() noexcept {}
    sleep(){}
};
```

文件中的 task_queue 存储的对象类型为 std::function<bool()>，可能是我的思维不够灵活，没太想出来要怎样使用这个 queue 才可以实现恢复对头的协程，所以我写了一个 my_queue。

```
static std::queue<std::coroutine_handle<>> my_queue;
```

剩下的就是实现 wait_task_queue_empty

```
void wait_task_queue_empty() {
    // TODO: block current thread until task queue is empty
    while (!my_queue.empty())
    {
        std::coroutine_handle<> h=my_queue.front();
        my_queue.pop();
        h.resume();
    }
}
```

参考资料：

1.课程主页的实验文档里列出的文章
2.v4 参考了 https://zhuanlan.zhihu.com/p/599053058
3.v5 参考了上面那篇文章和 https://godbolt.org/
4.关于标准库的部分用法参考自 CSDN，知乎，StackOverflow 上多篇文章