M2：协程库 (libco)

March 21, 2025

2025

我们在《操作系统》课程中学习了线程：只需要 create 和 join 两个 API，我们就能创建和管理线程。我们也知道，线程的创建和执行需要操作系统的帮助：程序就是状态机，如果执行死循环，这个状态机就 “卡死”了。那么，我们能在一个状态机内，实现多个状态机 (类似线程切换) 的效果吗?

> ⏰Soft Deadline: 2025 年 4 月 2 日 23:59:59
>
> 你需要首先阅读实验须知，其中包含了代码获取方法、提交方法、如何查看提交结果等。在命令行中 `git pull origin M2` 下载框架代码。

> ⚠️难度警告
>
> 在这个实验中，需要冷静下来调试你的代码——“看起来大概正确” 的代码很可能不能正常工作。如果发生奇怪的错误，不用着急，想一想上课介绍过的调试工具，想一想 “程序就是状态机”，然后开始理解出错的过程。请你相信你自己的能力，并且**尽最大可能遵守学术诚信，独立地设计测试用例、解决你遇到的问题**，这会使你的编程能力更上一个台阶。

1. 背景

设想我们实现了一个 SimpleC 的解释器 (参考上课时的非递归汉诺塔)，可以看成 “单步执行语句” 的状态机：

```
struct State {
    stack<Frame> frames;
    char *shared_memory;

    void next() {
        exec(frames.top().pc, shared_memory);
    }
};
```

而其中每一个 next 都是 “一小步”，除了系统调用之外，都可以很快返回，那么我们把 SimpleC 的解释器改造一下，不就可以实现，用一个线程模拟 “两个线程” 执行了吗：

```
struct State {
    stack<Frame> t1, t2;
    char *shared_memory;

    void next() {
        auto frames_chosen = random_choice({t1, t2});
        exec(frames_chosen.top().pc, shared_memory);
    }
};
```

那么，我们能不能更进一步，在一个线程里通过 malloc 创建若干份堆栈，然后在多个线程之间主动地切换呢？有些语言的确允许我们这么做，例如我们的 “操作系统” 模型：

> 我们实现一个 “模拟” 的、能够支持单个应用程序执行 (提供 read, write, spawn 三个 “极简“ 系统调用) 的操作系统模型，帮助理解操操作系统的工作原理。把待运行的程序 (hello.py, proc.py 等) 想象成状态机、操作系统想象成状态机的管理者，就不难理解操作系统是什么了。

例如我们的操作系统模型实现借助了 generator object：

```
def positive_integers():
    i = 0
    while i := i + 1:  # 死循环
        yield i  # "return" i，但之后程序可以从这里开始继续执行

def is_prime(i):
    return i >= 2 and \
        not any(i % j == 0 for j in range(2, i))

# 所有素数的顺序集合；注意其中的 positive_integers 是“死循环”
primes = (i for i in positive_integers() if is_prime(i))
```

调用 generator function 会创建一个闭包 (closure)，而不是直接陷入死循环：

```
>>> positive_integers()
<generator object positive_integers at 0x10e4d01c0>
>>> primes
<generator object <genexpr> at 0x10e4bcc80>
```

闭包的内部会存储所有局部变量和 PC，遇到 yield 时执行流返回，但闭包仍然可以继续使用，就好像 `stack<Frame>` 还被保留下来，随时可以继续执行。

2. 实验描述

> 📓实验要求：使用一个操作系统线程实现主动切换的多个执行流
>
> 在这个实验中，我们实现轻量级的用户态~~携谐~~协程 (coroutine，“协同程序”)，也称为 green threads、user-level threads，可以在一个不支持线程的操作系统上实现共享内存多任务并发。即我们希望实现 C 语言的 “函数”，它能够：
>
> - 被 `co_start()` 调用，从头开始运行；
>   - 在运行到中途时，调用 `co_yield()` 被 “切换” 出去；
>   - 稍后有其他协程调用 `co_yield()` 后，选择一个先前被切换的协程继续执行。
>
> 协程这一概念最早出现在 Melvin Conway 1963 年的论文 “Design of a separable transition-diagram compiler“，它实现了 “可以暂停和恢复执行” 的函数。

2.1 Coroutine APIs

实现协程库 `co.h` 中定义的 API：

```
struct co *co_start(const char *name, void (*func)(void *), void *arg);
void       co_yield();
void       co_wait(struct co *co);
```

协程库的使用和线程库非常类似：

1. `co_start(name, func, arg)` 创建一个新的协程，并返回一个指向 `struct co` 的指针 (类似于 `pthread_create`)。
   - 新创建的协程从函数 `func` 开始执行，并传入参数 `arg`。新创建的协程不会立即执行，而是调用 `co_start` 的协程继续执行。
   - 使用协程的应用程序不需要知道 `struct co` 的具体定义，因此请把这个定义留在 `co.c` 中；框架代码中并没有限定 `struct co` 结构体的设计，所以你可以自由发挥。
   - `co_start` 返回的 `struct co` 指针需要分配内存。我们推荐使用 `malloc()` 分配。

2. `co_wait(co)` 表示当前协程需要等待，直到 `co` 协程的执行完成才能继续执行 (类似于 `pthread_join`)。
   - 在被等待的协程结束后、 `co_wait()` 返回前， `co_start` 分配的 `struct co` 需要被释放。如果你使用 `malloc()`，使用 `free()` 释放即可。
   - 因此，每个协程只能被 `co_wait` 一次 (使用协程库的程序应当保证除了初始协程外，其他协程都必须被 `co_wait` 恰好一次，否则会造成内存泄漏)。
3. `co_yield()` 实现协程的切换。协程运行后一直在 CPU 上执行，直到 `func` 函数返回或调用 `co_yield` 使当前运行的协程暂时放弃执行。 `co_yield` 时若系统中有多个可运行的协程时 (包括当前协程)，你应当随机选择下一个系统中可运行的协程。
4. `main` 函数的执行也是一个协程，因此可以在 `main` 中调用 `co_yield` 或 `co_wait`。 `main` 函数返回后，无论有多少协程，进程都将直接终止。

2.2 使用协程库

下面是协程库使用的一个例子，创建两个 (永不结束的) 协程，分别打印 `a` 和 `b`。由于 `co_yield()` 之后切换到的协程是随机的 (可能切换到它自己)，因此你将会看到随机的 `ab` 交替出现的序列，例如 `ababbabaaaabbaa...`

```c
#include <stdio.h>
#include "co.h"

void entry(void *arg) {
    while (1) {
        printf("%s", (const char *)arg);
        co_yield();
    }
}

int main() {
    struct co *co1 = co_start("co1", entry, "a");
    struct co *co2 = co_start("co2", entry, "b");
    co_wait(co1); // never returns
    co_wait(co2);
}
```

当然，协程有可能会返回，例如在下面的例子 (测试程序) 中，两个协程会交替执行，共享 `counter` 变量：

```c
#include <stdio.h>
#include "co.h"

int count = 1; // 协程之间共享

void entry(void *arg) {
    for (int i = 0; i < 5; i++) {
        printf("%s[%d] ", (const char *)arg, count++);
        co_yield();
    }
}

int main() {
    struct co *co1 = co_start("co1", entry, "a");
    struct co *co2 = co_start("co2", entry, "b");
    co_wait(co1);
    co_wait(co2);
    printf("Done\n");
}
```

正确的协程实现应该输出类似于以下的结果：字母是随机的 (`a` 或 `b`)，数字则从 `1` 到 `10` 递增。

```
b[1] a[2] b[3] b[4] a[5] b[6] b[7] a[8] a[9] a[10] Done
```

从 “程序是状态机” 的角度，协程的行为理解起来会稍稍容易一些。首先，所有的协程是共享内存的——就是协程所在进程的地址空间。此外，每个协程想要执行，就需要拥有**独立的堆栈和寄存器** (这一点与线程相同)。一个协程的寄存器、堆栈、共享内存就构成了当且协程的状态机执行，然后：

- `co_start` 会在共享内存中创建一个新的状态机 (堆栈和寄存器也保存在共享内存中)，仅此而已。新状态机的 `%rsp` 寄存器应该指向它独立的堆栈，`%rip` 寄存器应该指向 `co_start` 传递的 `func` 参数。根据 32/64-bit，参数也应该被保存在正确的位置 (x86-64 参数在 `%rdi` 寄存器，而 x86 参数在堆栈中)。`main` 天然是个状态机，就对应了一个协程；
- `co_yield` 会将当前运行协程的寄存器保存到共享内存中，然后选择一个另一个协程，将寄存器加载到 CPU 上，就完成了 “状态机的切换”；
- `co_wait` 会等待状态机进入结束状态，即 `func()` 的返回。

2.3 协程和线程

协程和线程的 API 非常相似。例如课堂线程库中提供的

```
void create(void (*func)(void *));
void join();
```

刚好对应了 `co_start` 和 `co_wait` (`join` 会在 `main` 返回后，对每个创建的线程调用 `pthread_join`，依次等待它们结束，但我们的协程库中 `main` 返回后将立即终止)。唯一不同的是，线程的调度不是由线程决定的 (由操作系统和硬件决定)，但协程除非执行 `co_yield()` 主动切换到另一个协程运行，当前的代码就会一直执行下去。

协程会在执行 `co_yield()` 时主动让出处理器，调度到另一个协程执行。因此，如果能保证 `co_yield()` 的定时执行，我们甚至可以在进程里实现线程。这就有了操作系统教科书上所讲的 “用户态线程”——线程可以看成是每一条语句后都 “插入” 了 `co_yield()` 的协程。这个 “插入” 操作是由两方实现的：操作系统在中断后可能引发上下文切换，调度另一个线程执行；在多处理器上，两个线程则是真正并行执行的。

协程与线程的区别在于协程是完全在应用程序内 (低特权运行级) 实现的，不需要操作系统的支持，占用的资源通常也比操作系统线程更小一些。协程可以随时切换执行流的特性，用于实现状态机、actor model, goroutine 等。在实验材料最前面提到的 Python/Javascript 等语言里的 generator 也是一种特殊的协程，它每次 `co_yield` 都将控制流返回到它的调用者，而不是像本实验一样随机选择一个可运行的协程。

3. 正确性标准

实验要求：将所有代码实现在 co.c 中

不要修改 `co.h`。Online Judge 在评测时仅拷贝你的 `co.c` 文件。`struct co` 可以直接在 .c 中定义 (头文件不需要给出具体的结构体定义)。好消息是，我们已经为大家提供了基础测试用例，测试用例有两组：

1. (Easy) 创建两个协程，每个协程会循环 100 次，然后打印当前协程的名字和全局计数器 `g_count` 的数值，然后执行 `g_count++`。
2. (Hard) 创建两个生产者、两个消费者。每个生产者每次会向队列中插入一个数据，然后执行 `co_yield()` 让其他 (随机的) 协程执行；每个消费者会检查队列是否为空，如果非空会从队列中取出头部的元素。无论队列是否为空，之后都会调用 `co_yield()` 让其他 (随机的) 协程执行。

执行 `make test` 会在 x86-64 和 x86-32 两个环境下运行你的代码——如果你看到第一个测试用例打印出数字 `X/Y-0` 到 `X/Y-199`、第二个测试用例打印出 `libco-200` 到 `libco-399`，说明你的实现基本正确；否则请调试你的代码。
Online Judge 上会运行类似的测试 (也会在 x86-64 和 x86-32 两个平台上运行)，但规模可能稍大一些。你可以假设：

1. 每个协程的堆栈使用不超过 64 KiB；
2. 任意时刻系统中的协程数量不会超过 128 个 (包括 `main` 对应的协程)。协程 `wait` 返回后协程的资源应当被回收——我们可能会创建大量的协程执行-等待-销毁、执行-等待-销毁。因此如果你的资源没有及时回收，可能会发生 Memory Limit Exceeded 问题。

还需要注意的是，提交的代码不要有任何多余的输出，否则可能会被 Online Judge 判错。

☕希望在本地保留调试信息?

如果你希望在本地运行时保留调试信息并且不想在提交到 Online Judge 时费力地删除散落在程序中的调试信息，你可以尝试用一个你本地的宏来控制输出的行为，例如

```
#ifdef LOCAL_MACHINE
    #define debug(...) printf(__VA_ARGS__)
#else
    #define debug()
#endif
```

然后通过增加 `-DLOCAL_MACHINE` 的编译选项来实现输出控制——在 Online Judge 上，所有的调试输出都会消失。

4. 实验指南

☕感觉有点无从下手?

不要慌

虽然这个实验有点难

以前的实验都是有明确目标的，比如 OJ 题给定输入和输出。但这次不一样，我们要 hack C 语言运行时的行为——写一个函数 "切换" 到另一个函数执行。听起来就无从下手。

但是依然不要慌：程序就是状态机，总有办法的。让我们一起分析这个问题。

4.1. 编译和运行

我们的框架代码的 `co.c` 中没有 `main` 函数——它并不会被编译成一个能直接在 Linux 上执行的二进制文件。编译脚本会生成共享库 (shared object, 动态链接库) `libco-32.so` 和 `libco-64.so`：

```
$ file libco-64.so
libco-64.so: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), ...
```

共享库可以有自己的代码、数据，甚至可以调用其他共享库 (例如 libc, libpthread 等)。共享库中全局的符号将能被加载它的应用程序调用。共享库中不需要入口 (`main` 函数)。我们的 Makefile 里已经写明了如何编译共享库：

```
$(NAME)-64.so: $(DEPS) # 64bit shared library
	gcc -fPIC -shared -m64 $(CFLAGS) $(SRCS) -o $@ $(LDFLAGS)
```

其中 `-fPIC -fshared` 就代表编译成位置无关代码的共享库。除此之外，共享库和普通的二进制文件没有特别的区别。虽然这个文件有 `+x` 属性并且可以执行，但会立即得到 Segmentation Fault (可以试着用 gdb 调试它)。

❓ 奇怪的知识增加了

能不能得到既可以运行又可以动态加载的库文件?

是否可以让共享库同时作为命令行工具和动态链接库? 这并不是一个 "标准" 的操作系统——可执行文件和共享库有着不同的假设。但这件事的确可以办到！一个典型的例子是进程执行的第一条指令所在的 `/lib64/ld-linux-x86-64.so.2`：

$ /lib64/ld-linux-x86-64.so.2

Usage: ld.so [OPTION]... EXECUTABLE-FILE [ARGS-FOR-PROGRAM...] You have invoked `ld.so`, the helper program for shared library executables. This program usually lives in the file `/lib/ld.so`, and special directives in executable files using ELF shared libraries tell the system's program loader to load the helper program from this file. This helper program loads the shared libraries needed by the program executable, prepares the program to run, and runs it...

有兴趣的同学可以 STFW 这是如何做到的。

4.2. 使用协程库

为了降低大家 STFW/RTFM 的难度，我们提供了一组协程库的测试用例 (`tests/` 目录下)，包含了编译和运行所需的脚本，其中编译的编译选项是

```
gcc -I.. -L.. -m64 main.c -o libco-test-64 -lco-64
gcc -I.. -L.. -m32 main.c -o libco-test-32 -lco-32
```

注意到 `-I` 和 `-L` 选项的使用：

- `-I` 选项代表 include path，使我们可以 `#include <co.h>`。你可以使用 `gcc --verbose` 编译看到编译器使用的 include paths。
- `-L` 选项代表增加 link search path。
- `-l` 选项代表链接某个库，链接时会自动加上 `lib` 的前缀，即 `-lco-64` 会依次在库函数的搜索路径中查找 `libco-64.so` 和 `libco-64.a`，直到找到为止。如果你将 `libco-64.so` 删除后用 strace 工具查看 gcc 运行时使用的系统调用，就能清晰地看到库函数解析的流程；

在运行时，使用 `make test` 直接运行，它执行的命令是：

```
LD_LIBRARY_PATH=.. ./libco-test-64
```

如果不设置 `LD_LIBRARY_PATH` 环境变量，你将会遇到 “error while loading shared libraries: `libco-xx.so`: cannot open shared object file: No such file or directory” 的错误。请 STFW 理解这个环境变量的含义。

至此，我们已经完成了共享库的编译，以及让一个 C 程序动态链接共享库执行。

> 小技巧：调试你的代码
>
> 直接运行 `./libco-test-64` 会遇到 No such file or directory 的问题；当然，只需要用 `make test` 就可以解决这个问题了。但如果想要调试代码？`gdb libco-test-64` 同样也会遇到共享库查找失败的问题。大家可以在终端中使用 `export` 将当前 shell 进程的 `LD_LIBRARY_PATH` 设置好，这样就可以无障碍地运行 `./libco-test-64` 了。

4.3. 实现协程：分析

下面我们先假设协程已经创建好，我们分析 `co_yield()` 应该如何实现。

首先，`co_yield` 函数不能正常返回 (return；对应了 ret 指令)，否则 `co_yield` 函数的调用栈帧 (stack frame) 会被摧毁，我们的 yield 切换到其他协程的想法就 “失败” 了。如果没有思路，你至少可以做的一件事情是**理解 `co_yield` 发生的时候，我们的程序到底处于什么状态**。而这 (状态机的视角) 恰恰是解开所有问题的答案。

不妨让我们先写一小段协程的测试程序：

```
void foo() {
  int i;
  for (int i = 0; i < 1000; i++) {
    printf("%d\n", i);
    co_yield();
  }
}
```

这个程序在 `co_yield()` 之后可能会切换到其他协程执行，但最终它依然会完成 1, 2, 3, … 1000 的打印。我们不妨先看一下这段代码编译后的汇编指令序列 (objdump)：

```
push   %rbp
push   %rbx
lea    <stdout>, %rbp
xor    %ebx, %ebx
sub    $0x8, %rsp
mov    %ebx, %esi
mov    %rbp, %rdi
xor    %eax, %eax
callq  <printf@plt>
inc    %ebx
xor    %eax, %eax
callq  <co_yield>  // <- 切换到其他协程
cmp    $0x3e8, %ebx
jne    669 <foo+0xf>
pop    %rax
pop    %rbx
pop    %rbp
retq
```

首先， `co_yield` 还是一个函数调用。函数调用就遵循 x86-64 的 calling conventions (System V ABI)：

- 前 6 个参数分别保存在 rdi, rsi, rdx, rcx, r8, r9。当然， `co_yield` 没有参数，这些寄存器里的数值也没有意义。
- `co_yield` 可以任意改写上面的 6 个寄存器的值以及 rax (返回值), r10, r11，返回后 `foo` 仍然能够正确执行。
- `co_yield` 返回时，必须保证 rbx, rsp, rbp, r12, r13, r14, r15 的值和调用时保持一致 (这些寄存器称为 callee saved/non-volatile registers/call preserved, 和 caller saved/volatile/call-clobbered 相反)。

换句话说，我们已经有答案了： `co_yield` 要做的就是**把 yield 瞬间的 call preserved registers “封存” 下来** (其他寄存器按照 ABI 约定， `co_yield` 既然是函数，就可以随意摧毁)，然后执行堆栈 (rsp) 和执行流 (rip) 的切换：

```
mov (next_rsp), $rsp
jmp *(next_rip)
```

因此，为了实现 `co_yield`，我们需要做的事情其实是：

1. 为每一个协程分配独立的堆栈；堆栈顶的指针由 `%rsp` 寄存器确定；
2. 在 `co_yield` 发生时，将寄存器保存到属于该协程的 `struct co` 中 (包括 `%rsp`)；
3. 切换到另一个协程执行，找到系统中的另一个协程，然后恢复它 `struct co` 中的寄存器现场 (包括 `%rsp`)。

4.4. 实现协程：数据结构

例如，参考实现的 `struct co` 是这样定义的：

```
enum co_status {
  CO_NEW = 1, // 新创建，还未执行过
  CO_RUNNING, // 已经执行过
  CO_WAITING, // 在 co_wait 上等待
  CO_DEAD,    // 已经结束，但还未释放资源
};

struct co {
  char *name;
  void (*func)(void *); // co_start 指定的入口地址和参数
  void *arg;

  enum co_status status;  // 协程的状态
  struct co *    waiter;  // 是否有其他协程在等待当前协程
  jmp_buf        context; // 寄存器现场 (setjmp.h)
  uint8_t        stack[STACK_SIZE]; // 协程的堆栈
};
```

看起来就像是在《计算机系统基础》中实现的上下文切换！我们推荐大家使用 C 语言标准库中的 `setjmp/longjmp` 函数来实现寄存器现场的保存和恢复。在《计算机系统基础》实验中，我们已经用汇编代码实现了这两个函数。没有好好做实验的同学，要加油补上啦！

4.5. 实现寄存器现现场切换

分配堆栈是容易的：堆栈直接嵌入在 `struct co` 中即可，在 `co_start` 时初始化即可。但麻烦的是如何让 `co_start` 创建的协程，切换到指定的堆栈执行。AbstractMachine 的实现中有一个精巧的 `stack_switch_call`（`x86.h`），可以用于切换堆栈后并执行函数调用，且能传递一个参数，请大家完成阅读理解 (对完成实验有巨大帮助)：

```
static inline void stack_switch_call(void *sp, void *entry, uintptr_t arg) {
  asm volatile (
#if __x86_64__
    "movq %0, %%rsp; movq %2, %%rdi; jmp *%1"
      : : "b"((uintptr_t)sp), "d"(entry), "a"(arg) : "memory"
#else
    "movl %0, %%esp; movl %2, 4(%0); jmp *%1"
      : : "b"((uintptr_t)sp - 8), "d"(entry), "a"(arg) : "memory"
#endif
  );
}
```

理解上述函数你需要的文档：GCC-Inline-Assembly-HOWTO。当然，这个文档有些过时，如果还有不明白的地方，gcc 的官方手册是最佳的阅读材料。

> **警告：堆栈对齐**
>
> x86-64 要求堆栈按照 16 字节对齐 (x86-64 的堆栈以 8 字节为一个单元)，这是为了确保 SSE 指令集中 XMM 寄存器变量的对齐。如果你的程序遇到了神秘的 Segmentation Fault (可能在某个 libc 的函数中)，如果你用 gdb 确定到 Segmentation Fault 的位置，而它恰好是一条 SSE 指令，例如
>
> ```
> movaps %xmm0,0x50(%rsp)
> movaps %xmm1,0x60(%rsp)
> ...
> ```
>
> 那很可能就是你的堆栈没有正确对齐。我们故意没有说的是，System V ABI (x86-64) 对堆栈对齐的要求，是在 “何时” 做出的——在 `call` 指令之前按 16 字节对齐，在 `call` 指令之后就不对齐了。一方面你可以暴力地尝试一下；如果你想更深入地理解这个问题，就需要读懂 `stack_switch_call`，以及 STFW 关于 ABI 对对齐的要求，或是查看编译出的汇编代码。

每当 `co_yield()` 发生时，我们都会选择一个协程继续执行，此时必定为以下两种情况之一 (思考为什么)：

1. 选择的协程是新创建的，此时该协程还没有执行过任何代码，我们需要首先执行 `stack_switch_call` 切换堆栈，然后开始执行协程的代码；
2. 选择的协程是调用 `yield()` 切换出来的，此时该协程已经调用过 `setjmp` 保存寄存器现场，我们直接 `longjmp` 恢复寄存器现场即可。

当然，上述过程描述相当的抽象；你可能会花一点时间，若干次试错，才能实现第一次切换到另一个协程执行——当然，这会让你感到非常的兴奋。之后，你还会面对一些挑战，例如如何处理 `co_wait`，但把这些难关一一排除以后，你会发现你对计算机系统 (以及 “程序是个状态机”) 的理解更深刻了。

> ⚠️ 用好 gdb
>
> gdb 是这个实验的好帮手，无论如何都要逼自己熟练掌握。例如，你可以试着调试一小段 setjmp/longjmp 的代码，来观察 setjmp/longjmp 是如何处理寄存器和堆栈的，例如是否只保存了 volatile registers 的值？如何实现堆栈的切换？堆栈式如何对齐的?
>
> ```
> int main() {
>     int n = 0;
>     jmp_buf buf;
>     setjmp(buf);
>     printf("Hello %d\n", n);
>     longjmp(buf, n++);
> }
> ```
>
> 无论你觉得你理解得多么 "清楚"，亲手调试过代码后的感觉仍是很不一样的。

4.6. 实现协程

> 非常难理解？坚持住！
>
> 没错，的确很难理解。如果你没有完成《计算机系统基础》中的 `setjmp/longjmp` 实验，你需要多读一读 `setjmp/longjmp` 的文档和例子——这是很多高端面试职位的必备题目。如果你能解释得非常完美，就说明你对 C 语言有了脱胎换骨的理解。
> `setjmp/longjmp` 的 "寄存器快照" 机制还被用来做很多有趣的 hacking，例如实现事务内存、在并发 bug 发生以后的线程本地轻量级 recovery 等等。
> `setjmp/longjmp` 类似于保存寄存器现场/恢复寄存器现场的行为，其实模拟了操作系统中的上下文切换。因此如果你彻底理解了这个例子，你们一定会觉得操作系统也不过如此——我们在操作系统的进程之上又实现了一个迷你的 "操作系统"。类似的实现还有 AbstractMachine 的native，它是通过 `ucontext.h` 实现的，有兴趣的同学也可以尝试阅读 AbstractMachine 的代码。

在参考实现中，我们维护了 "当前运行的协程" 的指针 (这段代码非常类似于操作系统中，为每一个 CPU 维护一个 "当前运行的进程")：

```
struct co *current;
```

这样，在 `co_yield` 时，我们就知道要将寄存器现场保存到哪里。我们使用的代码是

```
void co_yield() {
  int val = setjmp(current->context);
  if (val == 0) {
    // ?
  } else {
    // ?
  }
}
```

在上面的代码中， `setjmp` 会返回两次：

- 在 `co_yield()` 被调用时， `setjmp` 保存寄存器现场后会立即返回 `0`，此时我们需要选择下一个待运行的协程 (相当于修改 `current`)，并切换到这个协程运行。
- `setjmp` 是由另一个 `longjmp` 返回的，此时一定是因为某个协程调用 `co_yield()`，此时代表了寄存器现场的恢复，因此不必做任何操作，直接返回即可。

最后，框架代码里有一行奇怪的 `CFLAGS += -U_FORTIFY_SOURCE`，用来防止 `__longjmp_chk` 代码检查到堆栈切换以后报错 (当成是 stack smashing)。Google 的 sanitizer 也遇到了相同的问题。

4.7. 资源初始化、管理和释放

> 需要初始化?
>
> 如果你希望在程序运行前完成一系列的初始化工作 (例如分配一些内存)，可以定义 `__attribute__((constructor))` 属性的函数，它们会在 `main` 执行前被运行。我们在课堂上已经讲解过。

这个实验最后的麻烦是管理 `co_start` 时分配的 `struct co` 结构体资源。很多时候，我们的库函数都涉及到资源的管理，在面向 OJ 编程时，大家养成了很糟糕的习惯：只管申请、不管释放，依赖操作系统在进程结束后自动释放资源。但如果是长期运行的程序，这些没有释放但又不会被使用的**泄露** 资源就成了很大但问题，例如在 Windows XP 时代，桌面 Windows 是没有办法做到开机一星期的，一周之后机器就一定会变得巨卡无比。

管理内存说起来轻巧——一次分配对应一次回收即可，但协程库中的资源管理有些微妙 (但并不复杂)，因为 `co_wait` 执行的时候，有两种不同的可能性：

1. 此时协程已经结束 (`func` 返回)，这是完全可能的。此时， `co_wait` 应该直接回收资源。
2. 此时协程尚未结束，因此 `co_wait` 不能继续执行，必须调用 `co_yield` 切换到其他协程执行，直到协程结束后唤醒。

希望大家仔细考虑好每一种可能的情况，保证你的程序不会在任何一种情况下 crash 或造成资源泄漏。然后你会发现，假设每个协程都会被 `co_wait` 一次，且在 `co_wait` 返回时释放内存是一个几乎不可避免的设计：如果允许在任意时刻、任意多次等待任意协程，那么协程创建时分配的资源就无法做到自动回收了——即便一个协程结束，我们也无法预知未来是否还会执行对它的 `co_wait`，而对已经回收的 (非法) 指针的 `co_wait` 将导致 undefined behavior。C 语言中另一种常见 style 是让用户管理资源的分配和释放，显式地提供 `co_free` 函数，在用户确认今后不会使用时释放资源。

资源管理一直是计算机系统世界的难题，至今很多系统还受到资源泄漏、use-after-free 的困扰。例如，顺着刚才资源释放的例子，你可能会感觉 `pthread` 线程库似乎有点麻烦： `pthread_create()` 会修改一个 `pthread_t` 的值，线程返回以后资源似乎应该会被释放。那么：

- 如果 `pthread_join` 发生在结束后不久，资源还未被回收，函数会立即返回。
- 如果 `pthread_join` 发生在结束以后一段时间，可能会得到 `ESRCH` (no such thread) 错误。
- 如果 `pthread_join` 发生在之后很久很久很久很久，资源被释放又被再次复用 (`pthread_t` 是一个的确可能被复用的整数)，我不就 join 了另一个线程了吗？这恐怕要出大问题。

实际上，pthread 线程默认是 “joinable” 的。joinable 的线程只要没有 join 过，资源就一直不会释放。特别地，文档里写明：

> Failure to join with a thread that is joinable (i.e., one that is not detached), produces a “zombie thread”. Avoid doing this, since each zombie thread consumes some system resources, and when enough zombie threads have accumulated, it will no longer be possible to create new threads (or processes).

这就是实际系统中各种各样的 “坑”。在《操作系统》这门课程中，我们尽量不涉及这些复杂的行为，而是力图用最少的代码把必要的原理解释清楚。当大家对基本原理有深入的理解后，随着经验的增长，就会慢慢考虑到更周全的系统设计。