M2: 协程库(libco)

截止日期

Soft Deadline: 3月31日23:59:59

收到的作业

引子：操作系统中线程的实现

在我们学习编程的时候，我们早已默认一个“进程”——一个运行的程序——拥有独立的地址空间。因此进程里的变量 x 是这个进程所独享的。在之前的课程上，我们也学习了用 create() 和 join() 管理同一个进程地址空间里多个并发/并行执行的线程。

你可能稍有惊讶的是，线程并不是什么神秘的东西。我们只需要给“线程”分配一段堆栈，稍微用上一些hacking，我们自己也能在进程里实现多个能并发执行的线程。

背景

如果大家用过Python/Javascript，一定多多少少了解过生成器(generator)和yield，例如，我们可以定义以下“死循环”的函数：

```
def integers():
  i = 0
  while True:
    yield i # "output" i
    i += 1
```

这个函数可以生成所有的整数，从而用于其他类型的数据生成：

```
def is_prime(i):
  return i ⩾ 2 and True not in (i % j == 0 for j in range(2, i))

primes = (i for i in integers() if is_prime(i)) # 所有的素数
# 类型: <generator object <genexpr> at 0×7f142c14c9e8>
```

虽然 integers() 是个死循环，但 primes 却不会无限地执行下去。我们始终可以用 next(primes) 得到下一个素数，而 primes 内部的状态 i 又被神奇的保存了下来。

如果我们希望在C里实现这样的“函数”，它能够：

- 可以被调用，从头开始运行；
- 在运行到中途时被“切换”出去，并返回一个值；
- 稍后可以“恢复执行”，即回到被切换时的状态继续执行。

你仔细一想就会发现有些小困难：如果我们我们在函数运行的过程中yield时使用函数返回指令，函数的调用栈帧(stack frame)就被摧毁，从此以后函数内部的状态(具体来说，刚才的局部变量 i)就变为了undefined，它会被后续的函数调用覆盖。

如果我们能实现永久保存一个函数调用的堆栈，我们就有机会切换到另一个函数执行；这天然就有了并发的线程——当一个线程执行 yield arg 时，则调度调用它的线程执行。这就是协程(coroutine) (https://en.wikipedia.org/wiki/Coroutine)，即“协作的程序”。如果我们允许协程在任意时候被中断(并且切换到其他协程执行)，我们就得到了操作系统理论书中常说的“用户态线程”。

实验描述

获取实验代码与提交

本学期的所有代码(minilab, OSlab)都在同一个目录中完成。请参考代码获取与提交 (OS2019_Code)。

在你原先的os-workbench基础上，执行

```
git pull origin M2
```

将本实验的框架代码下载到本地。在 libco/ 下编译能得到动态链接库 libco-32.so 和 libco-64.so 。测试代码在 libco/tests 。

在这个实验中，我们实现轻量级的用户态~~携带~~协程 (coroutine)，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多线程的能力。

实验要求实现四个函数，分别是：

```c
typedef void (*func_t)(void *arg);
struct co;

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

这些函数的实现会被封装成一个动态链接库(就像我们调用 `printf`，其实是调用了glibc动态链接库中的函数)，这样就能被其他程序调用了。函数的具体功能如下：

1. 使用协程的程序会首先调用 `co_init` 完成一些必要的初始化。如果你的实现并不需要在启动时做任何初始化，你可以留下一个空的函数。
2. `co_start` 创建一个新的协程，并返回一个指针(动态分配内存)。我们的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😀。
3. `co_yield` 是指当前运行的协程放弃执行，并切换到其他协程执行。系统中可能有多个运行的协程(包括当前协程)。你可以随机选择下一个系统中可运行的协程。
4. `co_wait(thd)` 表示当前协程不再执行，直到 `thd` 协程的执行完成。我们规定，每个协程的资源在 `co_wait()` 等待结束后释放，因此每个协程只能被 `co_wait` 一次。更精确地说，每个协程必须恰好被 `co_wait` 一次，否则就会造成资源泄露。

资源管理和释放

很多时候，我们的库函数都涉及到资源的管理，例如 `co_start` 时，需要申请 `struct co` 的内存等。在面向OJ编程时，我们从来都是只管申请不管释放的——进程结束后，这些资源都会被自动释放。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。

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 `wait`。当然，如果本着对自己不负责任的态度，你也可以选择不回收协程的资源。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 `free` 函数，在用户确认今后不会使用时释放资源。

然后，你立即就会感觉 `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).

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。

让我们回到协程API。协程运行的函数就是普通的C函数，只不过其中可以调用 co_yield()：

```
void foo() {
  while (1) {
    printf("x");
    co_yield();
  }
}

void bar() {
  while (1) {
    printf("y");
    co_yield();
  }
}
```

例如这样两个函数被两个协程执行的话，将会打印出由x和y组成的字符串，例如 xxyxyyxy ...

```
int main() {
  co_init();
  co1 = co_start("t1", work1, arg1);
  co2 = co_start("t2", work2, arg2);
  co_wait(co1);
  co_wait(co2);
}
```

例如上面的代码创建了两个新的协程，并且等待这两个协程都执行结束。如果要实现generator，可以让 `yield()` 能返回一个指向协程堆栈上的指针，这个数据在协程结束前都会保证可用。

协程 vs. 线程

细心的你也许已经发现了，协程和课堂上讲解过的线程并没有很大的区别，回顾我们课堂上使用的线程库(`threads.h`):

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

刚好对应了 `co_start` 和 `co_wait` (`join` 多次调用 `pthread_join` 等待所有线程结束)。唯一不同的是，线程是完全并发执行的，但协程会一直运行，直到执行 `co_yield()` 为止，才会切换到另一个协程运行。

因此，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。

协程的应用

协程的最重要特性是可以保存自己的本地状态，因此也用作状态机、actor model等。Python/Javascript等语言里的generator也是一种特殊的协程，它每次 `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。

实验指导

不要慌。

看到这实验要求，你是不是感觉心都凉了？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。

不要慌，我们会一点一点来分析这个问题，然后你就会发现——其实也没那么难嘛。

运行库

和之前的实验(把源文件编译成一个二进制文件)不同，本实验把源文件编译成后缀名为 `.so` 的共享库(shared object):

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

注意到我们实现的是一组库函数(没有main函数)，库函数是被其他程序调用的——这个 .so 共享库将会和其他程序动态链接，从而使另一个程序能调用我们的协程库函数。

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。虽然这个文件有+x属性并且可以执行，但会立即得到Segmentation Fault。当然，你的确可以让你的共享库真正可以执行(并得到有意义的结果)，同时作为命令行工具和动态链接库，有兴趣的同学请STFW。

测试你的代码

我们已经提供了一组测试用例，测试你的线程库(64bit)。

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

请大家RTFM以下编译选项的含义： -I，-L，-l 。然后使用

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

运行。这里我们配置了 LD_LIBRARY_PATH 环境变量，至于为什么，也请STFW。弄清楚这些以后，你就不难搞定32位版本的测试。

框架代码中的协程是用函数调用实现的——co_create 时立即执行函数调用， yield() 则直接返回。这对于第一个测试用例来说是正确的；但第二个用例会出现死循环等待。

奇怪的 CFLAGS

框架代码里有一行 CFLAGS += -U_FORTIFY_SOURCE ，用来防止 __longjm-p_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 (https://github.com/google/sanitizers/issues/721)。系统中就是有这么多神坑，感谢可爱的助教小哥哥背锅。

问题分析

实现协程最关键的地方在于理解函数调用。例如我们写一个函数：

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

它会编译成以下汇编代码:

```
push   %rbp
push   %rbx
lea    <stdout>, %rbp
xor    %ebx, %ebx
sub    $0×8, %rsp
mov    %ebx, %esi
mov    %rbp, %rdi
xor    %eax, %eax
callq  <printf@plt>
inc    %ebx
xor    %eax, %eax
callq  <co_yield>
cmp    $0×3e8, %ebx
jne    669 <foo+0×f>
pop    %rax
pop    %rbx
pop    %rbp
retq
```

因此，如果在 `co_yield()` 时切换到另一个函数执行，我们至少需要保存:

- 通用寄存器，因为后续汇编代码执行会依赖当前的寄存器状态(例如循环变量保存在 `%ebx` 中，如果EBX寄存器数据丢失，我们就丢失了 `i` 的数值)。
- 栈帧上的数据。我们看到 `foo()` 把旧的EBX寄存器的值保存在了堆栈上；如果栈帧被摧毁，则旧的EBX寄存器值将永远无法找回。此外，如果我们有更多的变量无法像 `i` 一样存储在寄存器里，则这些数据是保存在堆栈上的。

因此实现协程的任务一下就变得明确了：在 `co_yield` 时

- 保存当前所有的寄存器值。可以使用 `pusha` 指令(还需要保存额外的寄存器，例如 `rsp`)，也可以使用C标准库中的 `setjmp` 函数，这个函数甚至帮我们把PC指针都保存下来了。
- 保存当前的栈帧——当然更简单的实现方法是，我们为每一个协程都分配一个独立的堆栈，这样协程切换就不用担心栈帧被销毁的问题了。

自己试试吧!

我们已经给出了足够的提示，所以如果你想挑战一下自己，不妨不要阅读后面的部分，适当地STFW解决问题，这也将在完成后给你更多的成就感。

为每个协程分配堆栈

分配堆栈是非常容易的——用 `malloc` 分配一段内存即可，甚至我们可以直接把协程的堆栈定义在数据区：

```
struct co {
   ...
  uint8_t stack[4096];
};

struct co coroutines[MAX_CO]; // 静态分配
```

那么怎么为协程切换堆栈呢？我们来看一段ACM-ICPC界广为流传的外挂代码。在蛮荒时代，很多赛区使用默认的Linux堆栈，这会导致在有10^6级数据时递归堆栈溢出，因此有了如下代码，可以把一段数据区的内存作为堆栈：

```
#define KB * 1024LL
#define MB KB * 1024LL
#define GB MB * 1024LL
char __stack[4 GB];
void *__stack_backup;

#if defined(__i386__)
  define SP "%%esp"
#elif defined(__x86_64__)
  #define SP "%%rsp"
#endif

void stackoverflow(long long level) {
  if (level % (1 MB) == 0) {
    printf("level = %d\n", level);
  }
  stackoverflow(level + 1);
}

int main() {
  // 堆栈外挂——启动
  asm volatile("mov " SP ", %0; mov %1, " SP :
                "=g"(__stack_backup) :
                "g"(__stack + sizeof(__stack)));

  stackoverflow(0);

  // 堆栈外挂——关闭
  asm volatile("mov %0," SP : : "g"(__stack_backup));
  // 如果不切换回堆栈，ret指令将会返回到不合法的地址(Segmentation Fault)
}
```

32位和64位

回顾我们要求大家编写在两个不同字长的x86平台上都能工作的代码。setjmp，longjmp 都是可移植的函数(它们内部实现很不一样，有兴趣的同学可以参考libc里的实现)，但堆栈切换需要为两个平台分别编写——上面的代码已经给了大家足够的提示，应该如何兼容两个不同的体系结构了。

学过《计算机系统基础》的你应该不难理解这段代码，并且开动脑筋就可以知道如何在 co_start 的时候，为协程开辟新的堆栈了。你需要的文档：GCC-Inline-Assembly-HOWTO (http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html)。仔细阅读这个文档，因为很多框架代码里都用到了内嵌汇编，你会受益的。

堆栈对齐

x86_64 ABI要求rsp(栈顶)寄存器总是16字节对齐——这是因为128bit SSE指令在非对齐的情况下可能返回错误的结果。那么怎么让你的堆栈正确对齐呢……?

实现协程

我们给大家推荐的方案是 `setjmp` / `longjmp` ，这样就不必再纠结内嵌汇编、应该保存哪些寄存器之类的事情了。首先，我们需要维护“当前运行的协程”：

```
struct co *current;
```

这其实已经非常类似于在操作系统里实现线程了——我们会为每一个CPU维护一个“当前运行的线程”。因此，在协程执行时调用 `co_yield` ：

```
void co_yield() {
  int val = setjmp(current→buf);
  if (val == 0) {
    // ???
  } else {
    // ???
  }
}
```

给大家的提示是， `setjmp` 会返回两次：

- `setjmp` 会立即返回，对应了 `yield()` 的执行，此时我们需要选择下一个协程，并切换到这个协程运行(通过 `longjmp` 实现)。
- `setjmp` 是由另一个 `longjmp` 返回的，此时我们什么都不需要做，只要返回就行了。

非常难理解?

没错，的确很难理解。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试试。

至于实现 `co_wait`，我们只需要在 `struct co` 中加入协程的状态即可。这个实验的确有点难，但完成之后的那种通透感也是十分畅快的。setjmp/longjmp的checkpoint和还原还被用来做很多有趣的hacking，例如实现事务内存(http://www.doc.ic.ac.uk/~phjk/GROW09/papers/03-Transactions-Schwinde-wolf.pdf)、在并发bug发生以后的线程本地轻量级recovery(https://doi.acm.org/10.1145/2451116.2451129)。