Lab3 ICS-kieraylab说明文档

理论部分

Problem1

被调用者通用寄存器有：%rbp、%rbx、%r12、%r13、%r14、%15

Problem2

```
naive_func:
endbr64
movq (%rsp), %rax
movq %rax, (%rdi)
xorq %rax, %rax
retq
```

Problem3

因为对于naive_fun的操作我们不需要在栈帧上开辟新的空间去存储局部变量或者其他

Problem4

考虑到被调用者保存寄存器在调用者函数运行时不会发生改变，因此我们可以不保存%rbx,%r12,%13,%14,%15，只需要将函数B即将返回时程序的调用者保存寄存器+帧指针(%rbp)记录下来（栈的内存可以通过偏移量访问）

Problem5

Python的try-catch语法规则：

try/except语句主要是用于处理程序正常执行过程中出现的一些异常情况，try中的代码块我们在使用时不确定是否发生错误，如果触发error则被except语句catch住进行异常处理，如果可以正常执行，则不会被catch住进入except语句

Problem6

我们在进行try的时候，用记录下当前的上下文环境（包括栈帧指针等），如果触发错误则将其抛回去并且返回记录的上下文环境，抛出去的错误被catch住从而进行异常处理

Problem7

pycharm结果如图：

```
D:\anaconda3\python.exe D:/PycharmProject/center_adapter/main.py
0
1
2
3
4
5
6
7
8
9
Traceback (most recent call last):
  File "D:\PycharmProject\center_adapter\main.py", line 6, in <module>
    print(next(g))
StopIteration

Process finished with exit code 1
```

Problem 8

对于generator，我们在调用generator之前记录当先上下文环境A,然后恢复到generator运行的环境B，然后执行generator函数，遇到yield语句则记录好当前环境B，然后恢复到环境A，这样的来回的上下文切换便能实现generator

Problem 9

调用者在try的时候记录好上下文，然后在generator设置error变量，每次用next(g)或者send(g， value)即调用生成器的时候通过检查error的值，如果有未被处理的异常则将其抛给generator的调用者，即被catch住然后调用者将处理异常

实验部分

test 1 2

首先我在做test1和test2中，需要去思考如何保存以及恢复，并且区分当前执行的是恢复后的返回还是仅仅在保存。在实现ctx_record时我**仅仅用 ctx去保存了(%rsp)返回地址，%rsp栈指针，以及%rbp帧指针**，然后将**返回值%rax设置为0**（以此作为和recover的区分）；然后在实现 __ctx_recover的时候,**我便将存好的值再赋值回去，同时将第二个参数的值赋值给返回值**(movq %rsi, %rax)对外伪造record式的返回并以返回值作为区分，从而实现上下文的保存与恢复

注：test1和2完成时还未修改__ctx_type的结构(orz),并且保存的寄存器无法支持后面test通过，但是并不影响1和2通过（显然）

test 3 4

在做test3和test4时，为了更方便搭建链表，我将ctx的结构改为如下（**用last指针指向上一个保存的上下文**）：

```
typedef struct Ctx{
    char a[80];
    struct Ctx *last;
}__ctx_type;
```

- **__eh_push()的实现：** 实际上我们的链表头__eh_list指向的是当前最新的上下文，我们在Push ctx（即当下最新的上下文）的时候需要将其**指向前一个上下文(last),然后更新链表头指向这个(ctx)**
- **__eh_pop()的实现：**pop的思路和push相反，我们首先将返回值设置为链表头的所指的上下文，然后更新链表头指向前一个上下文
- **try的实现：**受理论部分启发，在try的时候我们可以先用__ctx_record记录好当前上下文ctx,然后将其串入链表，并用error记录其返回值，如果**error==0则继续执行try中的代码块**，如果**error不为0则代表出现错误并且此时是recover的返回，则将执行catch的代码块**
- **catch的实现：**catch实际上只需要判断error是否为0，我们在try中加入if语句判断，如果error==0则继续运行，然后catch中放入else语句则进行异常处理
- **throw的实现：**throw函数中，**我们需要借用ctx_recover()函数跳回try中，并将error抛出**，其中__ctx_recover函数第一个参数是我们pop出的保存的最新的上下文
- **__eh_check_cleanup的实现**：在这一块中我们实际上是判断try是否正常运行(包括本身正常以及break或者continue的运行)，**如果error为0我们则将弹出先前放入异常栈中处理的最新上下文**

复盘一下就是说：我们弹出上下文有两种情况，一种是try正常执行，我们在析构函数中弹出并且recover回去，以error==0的形式返回；

另外一种是throw抛出异常error的返回，它将被catch住然后执行catch接下来的代码块

test 5 6

- **独立栈设计**（generator的初始化）：为了实现交替执行两个上下文环境，这里我们需要手动配置一个运行栈，**以保证generator初次运行的不是在主程序栈中record的上下文，而是我们给它配置的上下文**.观察main test5可知，我们实际上**想让recover到test5(1)**这个位置从而才能进入yield，所以我们初始设置(%rsp)=f（即test5),并且给它传入第一个参数arg,然后在设计%rsp和%rbp具体在栈中的位置，考虑到后面test5函数运行完返回地址出栈（%rsp）为空，然后%rsp往上移动8个字节，方便跳板函数的设计我就在初始化的时候%rbp和%rsp对应栈中的位置相差8，从而可以把跳板函数的地址压入(%rbp),然后最后然%rsp指向的地址即为(%rbp)便为我们想要它跳到的位置。

- **上下文切换实现**：（yield和send的极限拉扯...):实现了两个栈，我们就可以~~开心的~~反复横跳了，这里我们用this_gen指向当前运行的generator（也可以这样理解吧，把主程序视为一个generator),然后这两个generator在不同的栈中运行。在执行yield函数时，我用 int t = **ctx_record(this_gen->ctx)**,一个是记录当前generator的上下文，然后判断t是否为0来区别是record还是recover回来的，**如果t==0,则将目标生成器的调用者指向当前生成器，并且更新__this_gen为目标寄存器**，否则则返回对应的值；send函数执行同理；**我们通过不断更新__this_gen来实现是上下文的切换以及运行栈的切换**

- **异常error抛出+链表头为空的实现**：在最先设计独立栈的时候，我们压入了一个跳板函数的地址，从而实现yield对应的那个for循环结束后然后pop出test5(1)的返回地址，然后**让%rsp指向我们设计的函数地址，进入skip_throw(),**然后跳到throw(-2),此时我们需要判断链表头是否为空(为空代表仍然在我们设计的generator中)，然后我们需要将__this_gen切换回主程序对应的generator，然后重新throw（-2），这时我们便实现未处理异常地抛出并且清空链表

```
void throw(int error)
{
```

```
    if (error == 0)
        throw(ERR_THROW0);
    if(__this_gen->eh_list==NULL){
      __this_gen = __this_gen->caller;
     throw(ERR_GENEND);
      }

    __ctx_recover(__eh_pop(), error);

    assert(0); // shouldn't run here
}
void skip_throw(){ //跳板函数（跳回主程序generator的throw)
  throw(ERR_GENEND);
}
```

lab感受

感觉这个lab难度比之前高好多。。。有一种在北极寻找企鹅的无助感和无望感。。。。之前一直对函数调用栈帧的实现没搞懂，但是这个lab治好了我的精神内耗(orz)，让我在这个寒冷的期中季头脑疯狂一下。。。总的来说觉得这个lab认真做完确实对理解栈帧和函数调用还是有很大帮助！！！