Lab3_report

姓名：陈锐林 学号：21307130148

理论部分

Problem1:

%rbx %rbp %r12 %r13 %r14 %r15

Problem2:

```
endbr64
mov %rsp,(%rdi)
movq $0,%rax
retq
```

Problem3:

上课讲的 push rbp; mov rbp,rsp 是为了开新的栈帧，开了新的空间；

但既然调用的函数形式简单，没有再次调用其他复杂的函数，并且用不到栈来存储多余的参数的话，就没必要使用了。

Problem4:

仅保存栈指针和寄存器，不保存栈内存。

Problem5:

C++ try-catch 机制

```
try {
   // statement(s1)
} catch (ExceptionName e) {
   // statement(s2)
}
```

s1 处编写可能引发异常的语句，可以用 throw 抛出异常；

s2 处，针对 catch 捕捉到的异常，规定发生异常时进行接下来的处理。

Problem6:

如第一节所说，当遇到 try 语句时，可以保存下当前的上下文，接着进入 try 后的语句进行执行；可以预先设计错误检测机制，当检测到错误时，就回退到该 try 语句前；此时可以转而进入 catch 中的语句（正如 Problem5 中的 try-catch 机制）。

Problem7:

0
1
2
3
4
5
6
7
8
9

Problem8:

如 Problem7 中的例子，想要做到 generator 机制，在当退出执行的函数（如这这里的 gen 时）必须保存这时候函数执行到哪了；那么再下一次调用时就可以回到上次没执行的位置。那么联系第一节中上下文的概念，只要我们在退出所执行函数（如 gen）时保存该函数内的上下文，在从主函数到 generator 函数时保存主函数的上下文，并做到来回的复制和执行就可以了。

Problem9:

当在 generator 与主函数切换时，可以加入判断，如有一全局变量 error，在进入 generator 时赋为 0，若遇到了 throw 抛出的异常且未在 generator 中处理时，更改它的值；在进行 recover 到主函数时对此指标进行检测。

Lab 部分

1.上下文保存与恢复：

（1）完成内容：record 以及 recover 函数。

（2）具体实现：用汇编语言书写，record 函数将当下的 rbp rsp rip(即(rsp))，赋给 __ctx_type 类型的上下文；recover 相反，并且将 rsi 作为返回值返回；__ctx_type 类型已经经过重新封装，成员有 u64 的 rsp rbp rip 以及指向下一上下文的指针 struct node* next（node 为__ctx_type 的别名）。在汇编中，从(%rsp)到 16(%rdi)的赋值需要其他寄存器为中间载体（如 rdx），因为内存到内存的赋值是不允许的。

(3)实现思路：参考理论部分以及 kieraylab 里的指引；联系书上栈运行的图示。(图 3-25)

(4)问题：一开始我把 16(%rdi)误写成了 0x16(%rdi)，虽然这与结构体中我定义的位置并不一样，但 test1，test2 仍然不报错地通过了。但之后地 test3 和 test4 即使其他没有问题也运行不了，因此 debug 了很久；不知道为啥会出现 1/2 过了，3/4 却过不了的情况。

2.基于上下文回退的异常处理

（1）完成内容：try 和 catch 的宏定义；throw 函数；clean-up function；针对异常处理栈的 push 和 pop 操作。

（2）具体实现：try 即实现 record 当前上下文，若返回为 0，将上下文入栈则进行之后的操作；catch 则实现将上下文出栈；try 和 catch 可以拼接为一个 if-else 语句。throw 函数完成异常处理，从异常处理栈中弹出保存的前一条上下文并进行 recover 操作（第二个参数为 error）；clean-up function 的作用在于对 break 之类的情况进行改进，那么在 try 处可先对 error 赋值为-20（经观察，该值在其他情况下不会被赋给 error），clean-up 相关函数只要对 error 进行判断，若为-20 则为正常结束（break 类操作），于是把保存的上下文出栈，以免堆积影响到正确进行。push 操作让新加入的上下文 ctx 指向__this_gen->eh_list(异常处理栈的头,上一次保存的上下文),然后让__this_gen 指向 ctx 即可；pop 操作相反，在已保证异常处理栈中有内容的前提下，只要让__this_gen 向前指向稍早一条的上下文，弹出头条上下文就可以了。

（3）实现思路：try-catch 的宏定义即参考 TA 讲解时进行的处理，只有在对 error 提前赋-20 时是我想到反正 try 后面的语句会抛出新的 error，不如我这里先赋值了也无伤大雅，并且在看来 main 函数过后发现 throw 不会抛出-20 后，就决定这么做了；throw/push/pop 要实现的内容 kieraylab 里也给出了，我进行了一个文字转为代码的工作（但我一开始没有当成栈，而是当成队列；并且没有意识到__this_gen->eh_list 即最近保存的 ctx 是很愚蠢的）。

3.基于上下文切换的处理器

（1）完成内容：yield send generator 函数。

（2）具体实现：yield 函数 record 当前上下文到__this_gen 的 ctx，recover 到__this_gen 的 caller 的 ctx；send 函数在已有基础上 record 当前上下文（同上），设置__this_gen 为目标 gen 的 caller 并且恢复到目标 gen 的上下文；generator 函数完成首次 send 上下文的配置，将 stack 从大到小看，最大处为针对函数 f 返回后进行的抛出 ERR-GENEND 的函数的地址，往下一单位是开始时第一次进行函数 f 的入口，并且将 g 的 ctx 指向此处。generator 函数用到了其他两函数：一个是 initial，用于第一次 send 操作 recover 到这里，不直接用 f 因为无法传递参数，在这个函数里调用 g->f（参数为*stack，这里之前被设置为了一开始的参数 arg）；一个是 over 函数，用于调用标识函数返回，抛出错误，回到调用者。另外要修改 throw 函数，使栈空时返回上一级 caller 处理。

（3）思路：yield 和 send 也是根据文档内容写的代码，跟着 TA 走就对了！

然后 generator 函数这里主要是 stack 的利用和 arg 的保存，如下图：

stack（保存 arg）	省略的空间	stack+8176 initial 函数	stack+8184 over 函数

ctx->rsp

之后第一次 send 过后

stack（保存 arg）	省略的空间……	stack+8184 over 函数

ctx->rsp（随 send yield 变动）

函数真正结束后：

stack（保存 arg）	省略的空间	stack+8184 over 函数

ctx->rsp

Re.对函数进行包装，另外设函数，主要是出于 arg 的处理以及模拟栈的运行过程。

（4）遗留问题：在 send 函数中两次对 gen->error 进行判断，既然过程中 gen 不被改变，那么这么做的意义在哪呢，没想到，请助教赐教。

4. 其他杂谈

浅谈一下做这个 lab 的感受，可以无视 QAQ。首先，我觉得这是挺有意思的一个 lab（除了做起来太难受之外）；其次是可能比较陌生，刚开始做这个的时候刚学完第二章就云里雾里的，不是很懂栈帧的具体作用，再加上对 try-catch，generator（电导没上课 python 没用心学呜呜）不是理解得很透彻，让一切变得很困难，当然以上这些可能是个例也是我自己的问题；还有就是这几次 lab 的总体感觉，还是调试和修改有点困难，仍然没有成为 gdb 的熟练使用者（我自己要反思）；最后还是建议能多一丢丢的引导？像这次那个录屏前面没声音 and 最后说要点一下 arg 的处理也没说就增加了不小的难度。

最后，还真贴一只猫猫