ICS Lab3
实验部分

Kieray Lab*

Fudan-ICS

2022 年 10 月 20 日

本文档是 ICS Kieray Lab 指导文档，即 Lab3 的实验部分。Kieray Lab 的代码量较少，但可能需要一些前置知识。因此，请务必在进行 Kieray Lab 前完成 Lab3 的理论部分，否则你可能难以理解实验需要实现的内容。

在 Kieray Lab 中，我们将实现程序上下文[1]的保存与恢复，并在此基础上，通过上下文回退实现异常处理[2]机制，通过上下文切换实现生成器[3]机制。

1 文件结构与运行方法

Kieray Lab 发布包包含下列内容

- `Makefile` 为编译脚本。你可以在终端执行`make`来编译你的代码。如果编译成功，你可以执行`./main`来运行测试程序，并观察测试结果。如有需要，也可以使用`gdb main`来调试程序。执行`make clean`可以删除所有编译出的文件。
- `main.c` 为测试代码。其中包含若干个测试，测试将调用你编写的代码，并检查其是否正确工作。除出于调试目的的临时修改外，你**不能**修改这个文件的内容。
- `happy.h` 为实验通用头文件。你**需要**完成其中的 TODO 部分，且**不能**修改 TODO 以外的部分。

*~~网上搜不到就对子~~

[1] context，代码中将简称 ctx

[2] exception handling，代码中将简称 eh

[3] generator，代码中将简称 gen

- `ctx.S` 为上下文保存恢复的汇编代码。你**需要**完成其中的 TODO 部分。如有需要，你也可以修改文件的其他部分。
- `gen.c` 为 generator 的 C 代码。你**需要**完成其中的 TODO 部分。如有需要，你也可以修改文件的其他部分。

不禁止在 C 代码里使用内联甚至裸汇编，但我们保证这不是必须的。如无特殊情况，原则上不允许自行添加新文件。

2 上下文保存与恢复

在理论部分约定，即上下文只包括栈内存和寄存器的基础上，我们将对上下文的保存和恢复提出进一步限制，通过仅以约定的方式保存和恢复上下文以减少需要保存的内容。

考虑到栈上保存着函数的局部变量、调用关系等重要数据，一般地，在 record 操作[4]中，我们需要保存函数的栈内存。但是，假如我们约定任意调用 record 的函数**永远不会在其返回之后使用 record 保存的上下文**[5]，即 record 所保存的上下文的生命周期不超出其调用者的生命周期，那么不难看出，record 可以**只保存栈指针而不保存栈内存**。这是因为，除调用 record 的函数的栈帧和各上级栈帧以外，剩余的都是栈上的空余空间，这些空间中的数据本来就不需要被保存。而对于已有的各级栈帧，它们在调用 record 的函数返回之前都不会被释放，因此只会被对应的函数改变，这是完全可控的，并不会引发意料之外的结果，也就不需要保存。

试参考下面代码

```
1 funcA:
2     x = 0
3     record
4     x = 1
5     call funcB
6
7 funcB:
8     recover
```

显然，如果`x`被保存在寄存器中，`funcB`执行 recover 后，`funcA`从 record 处继续执行时，`x`的值为 0。而如果`x`被保存在栈上，`funcA`从 record 处继续执行时，因为我们没有保存栈上的数据，`x`的值将会是 1。

[4]我们称保存当前的上下文为 record 操作，恢复保存的上下文为 recover 操作

[5]使用即指通过 recover 恢复到该上下文

这似乎会导致程序执行出现一些问题。但是容易发现，这种混乱的影响范围仅限于`funcA`的局部变量。更准确的说，是`funcA`在 record 到 recover 的执行过程中改变过的局部变量，而局部变量只对`funcA`自身可见，因此是完全可控的。只要规定 **record 之后修改过的局部变量的值在 recover 之后都是未知的**，并要求如`funcA`这样的 record 的调用者在编写时考虑到这种影响，即可解决未保存栈上数据导致的问题。

此外，我们注意到由 recover 恢复上下文之后的返回对外表现为 record 的“返回”。因此，有必要引入区分 record 在保存上下文之后的返回和恢复上下文之后的返回的机制。容易发现，只要为 record 增加一个返回值，并约定 record 返回零时表示保存上下文之后的返回，返回非零时表示恢复上下文之后的返回，在 record 和 recover 中设置不同的返回值，即可实现对这两者的区分。

亦可参考下面代码

```
if (record() == 0):
    print("Record!")
    recover()
else:
    print("Recover!")
```

该代码将先后输出`Record!`和`Recover!`。

在`happy.h`中，我们定义了`__ctx_record`和`__ctx_recover`两个函数，前者执行 record 操作，用于保存上下文，后者执行 recover 操作，用于恢复上下文[6]。

`__ctx_record`定义为`int __ctx_record(__ctx_type*)`。该函数需要实现的功能为：**将当前的上下文保存到参数指定的内存地址，然后返回 0**。请注意，如前面的讨论，执行 recover 时将“伪造”record 的返回，此时`__ctx_record`会“返回”非零值。

`__ctx_recover`定义为`void __ctx_recover(__ctx_type*, int)`。该函数需要实现的功能为：**从第一个参数指定的内存地址处取得先前 record 的上下文，恢复该上下文并将第二个参数作为返回值返回**。请注意，如前面的讨论，恢复上下文后，程序已经回到了先前执行 record 时候的状态，因此 recover 并不会以其自身的身份返回，其返回对外表现为 record 的返回。

`__ctx_record`和`__ctx_recover`中都包含了`__ctx_type*`类型的参数。`__ctx_type`定义在`happy.h`中，**被用于存储保存的上下文**。Kieray Lab 发布包中提供的默认定义为`typedef char __ctx_type[128]`，即只是为保存上下文提供了 128 字节空间。你可以根据自己需要保存的内容调整空间的

[6]设计参考了 libc 中的 setjmp 和 longjmp。

大小，也可以将其改为一个结构体以描述更丰富的内部结构。**为方便后续实现 eh 和 gen，强烈建议你的 ___ctx__type 中有至少能够保存一个指针的空余空间。**

`__ctx_record`和`__ctx_recover`的实现**需要**由你自己完成。显然，C 语言是不能完成 record&recover 这样的魔法操作的[7]，因此你需要在`ctx.S`中按照 TODO 的提示编写相应的汇编代码。

> 一些与汇编书写相关的提示：按照 x86-64 调用约定，rdi 和 rsi 分别为函数调用的第一二个参数，rax 为函数调用的返回值。在默认情况下，gcc 使用 AT&T 汇编语法。如果你想要使用 Intel 汇编语法，可以在`ctx.S`的开头加上一句`.intel_syntax noprefix`。

如果一切顺利，在完成本节内容后，你将能通过 test1 和 test2。

3 基于上下文回退的异常处理

在上下文保存与恢复的基础上，我们可以实现一些有趣的东西，比如 try-catch。实现 try-catch 的思想是非常朴素的：只要在执行 try 时，保存当前的上下文，并记录到某个地方。然后在执行过程中，如果触发了 throw，就恢复先前返回的上下文，并通过 record 返回值的方式告知程序，使之进入 catch 分支。

不妨参考下面代码，其中已经初具 try-catch 的雏形

```
if (record() == 0):
    do try...
else:
    do catch...

def throw:
    recover()
```

借助 C 语言中宏的一些技巧，我们可以将上面的 if-else 包装为 try-catch 语句。但在深入探讨实现 try-catch 的技术细节之前，我们先来考虑一下 try-catch 嵌套的可能性，如下面代码所示

```
try:
    do...
    try:
        do...
```

[7] **禁止使用 setjmp 或 ___builtin__setjmp 等现成函数**，但你也许可以参考它们的实现

```
    catch:
        do...
catch:
    do...
```

在发生 try-catch 嵌套时，程序中会同时存在许多个 try 记录的上下文。在执行 throw 操作时，我们需要匹配最近的 catch 语句，即恢复到最后一个上下文。容易看出，这实际上就是一个**栈**的结构，我们将其称为**异常处理栈**。对于每个 try 操作，它记录当前的上下文，并将其加入异常处理栈。对于每个 throw 操作，它弹出异常处理栈中的栈顶元素，并以此恢复到最近一次 try 操作的上下文，转而执行 catch 操作。当然，清理也是必须的，对于每个正常完成[8]的 try 操作，我们需要弹出异常处理栈中的栈顶元素，使之恢复到 try 操作之前的状态。

在 Kieray Lab 中，我们将使用单向链表实现异常处理栈。在`gen.c`中，已经定义了一个链表头`__this_gen->eh_list`[9]，通过先前在`__ctx_type`中预留的一指针空间，你可以将 try 保存的上下文串入链表中，使链表头能够指向最近一次 try 操作保存的上下文。

异常处理栈的入栈和出栈操作分别定义为`__eh_push`和`__eh_pop`，你**需要**在`gen.c`中补全它们。`__eh_push`的参数和`__eh_pop`的返回值均为指向保存的上下文的指针，请注意区分链表节点的指针和上下文的指针，两者可能会相差一个偏移量。

你还**需要**补全 try、catch 和 throw 的代码。其中，`throw`定义在`gen.c`中。如前所述，它从异常处理栈中弹出最近的 try 操作保存的上下文，并执行 recover 操作，错误 ID[10]亦被通过 recover 传递。请注意 record 约定 0 表示保存上下文之后的返回，因此作为恢复上下文之后的返回，recover 操作传递的返回值不能为 0，否则将导致无法区分。Kieray Lab 的发布包中，已经在`throw`的开头提供了检查代码，无需大家再行考虑这一问题。

`try`和`catch`以宏的形式定义在`happy.h`中。如前所述，你可以考虑以 if-else 的形式实现 try-catch 操作。你还**需要**将 recover 传递的错误 ID 保存在名为 error 的变量中（我们已经给出了定义），以供 catch 中的代码使用。在实现 try-catch 时，请不要忘记为正常完成的 try 操作弹出先前放入异常处理栈中的上下文。一般地，你可以在 else 语句之前添加相关代码。考虑到 try 过程中可能执行的 break/return 等操作，Kieray Lab 的发布包中为 error 添加了类似于析构函数的 clean-up function[11]，该函数将在 error 的作

[8] 过程中没有执行过 throw

[9] 目前只要直接使用即可，generator 相关的概念会在下一节讲解

[10] throw 的参数，将被提供给 catch 部分的代码

[11] 可参考https://gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html

用域终结时被自动调用，也许你可以借助这套机制捕获 try 过程中执行的 break/return 等操作，保证异常处理栈能够被恢复原样。

作为总结，我们最后来看一下这套 try-catch 的设计是否符合第一节中所讨论的约束条件。显然，在函数返回时，其中涉及的所有 try 操作必须已经完成或触发 catch，这保证了相关的上下文已经被从异常处理栈中弹出，因此符合函数返回后，其中使用 record 保存的上下文不会再被使用的要求。在发生异常时，默认 try 作用域内修改过的所有局部变量的值都是不可靠的是合理的，这也符合 record 之后修改过的局部变量的值在 recover 之后都是未知的规定。

如果一切顺利，完成本节内容后，你将通过 test3 和 test4。你也可以自己玩一玩

```
int main()
{
    try
    {
        printf("hello\n");
        throw(2333);
        printf("shouldn't run here\n");
    }
    catch
    {
        printf("%d\n", error); // should print 233
    }
    printf("bye\n");
    return 0;
}
```

4 基于上下文切换的生成器

上下文保存与恢复最经典的应用莫过于上下文切换了。对于生成器的场景，一个非常显然的思路是，我们可以通过简单的 record&recover 组合来实现 generator 之间的切换。试考虑下面代码

```
ctx a, b;

funcA:
    for i in 0..10:
        print("A", i);
        if (record(&a) == 0)
```

```
7               recover(&b);
8
9   funcB:
10      for i in 0..10:
11          print("B", i);
12          if (record(&b) == 0)
13              recover(&a);
```

容易看出，如果这段代码已经跑起来了，它可以交替执行 A、B 两个循环，只要在此基础上包装出 yield 和 send 函数，即可达到与 Python 中 generator 一致的效果。但我们首先要解决一个问题：它是怎么跑起来的？

请注意我们在第一节中提出的约定：函数返回之后，不能再使用其调用 record 保存的上下文。因此，要想使`funcA`中引用的`ctx b`有效，必须保证`funcB`不返回，这似乎只可能发生在`funcB`调用了`funcA`的情况下。但反过来，要想使`funcB`中引用的`ctx a`有效，也必须保证`funcA`不返回，似乎只可能发生在`funcA`调用了`funcB`的情况下。

上面的分析仿佛引出了一个悖论，即我们所设计的 generator 看似可以正确运行，但使之正确运行的状态似乎是不可达到的。事实上，这一悖论只会在`funcA`和`funcB`使用同一个调用栈的情况下成立。假如我们为`funcA`和`funcB`分配两个不同的栈空间，那么执行`funcA`时，`funcB`的栈帧保留在它自己的栈空间上，与`funcA`所在的栈空间并无关联，也就不需要调用关系来保证`funcB`不返回了。

显然，为了给 generator 设置独立的栈空间，初次运行 generator 时不能使用直接调用的方式。这是因为，如果你直接调用 generator 对应的函数，使用的仍是主程序的栈空间，那么 generator 中在执行 record 操作时，记录的栈指针指向的也仍是主程序的栈空间，不能达到切断调用关系的效果。为了让 generator 能够在独立的栈空间上运行，**需要手动配置初始的上下文**，即 generator 初次运行时 recover 的上下文并不是 record 记录的，而是在创建 generator 的代码中直接配置的。为 generator 配置初始的上下文并不困难，你只要将其中对应于栈指针和返回地址的项填充为合适的数值即可[12]。

在`gen.c`中，我们已经定义了`struct gen`和`struct gen* __this_gen`。`struct gen`用于保存一个 generator 相关的状态信息，`__this_gen`则被设计为指向当前所在的 generator 的`struct gen`结构体。为了方便起见，我们将主程序也视为一个 generator，为其分配了一个`struct gen`结构体，并将`__this_gen`的初始值设为指向主程序的`struct gen`结构体。

`struct gen`中各成员的含义如下

[12]请注意栈帧（从返回地址开始）的地址必须 16 字节对齐。我尚未在官方文档中找到相关的指示，但不满足该对齐会导致 Segmentation fault。

- `error` 如果 generator 中发生了错误[13]，且该错误未在 generator 内部通过 try-catch 被处理，就会设置这一项。通过在 send 操作时检查 struct gen 的 error 字段，即可将 generator 内部未处理的异常向外抛出，可参考 Kieray Lab 发布包中已经给出的部分 send 函数代码。

- `data` 因为借助 recover 传递的数据不能为 0，我们选择借助 struct proc 的 data 字段来传递 yield/send 的数据。可参考 Kieray Lab 发布包中已经给出的部分 send 和 yield 函数代码。

- `stack` generator 的独立栈空间的基地址指针。在创建 generator 时分配，销毁 generator 时释放。可参考 Kieray Lab 发布包中已经给出的部分 generator 和 genfree 函数代码。

- `ctx` generator 的上下文。在执行 send 操作时，会基于此上下文进行 recover，以执行 generator 的代码。在执行 yield 操作时，会将 generator 当前的上下文 record 到这里。

- `caller` generator 的调用者。在执行 send 操作时，会将目标 generator 的 caller 设置为当前 generator。在执行 yield 操作时，会根据 caller 字段 recover 到调用者的上下文。

- `eh_list` generator 的异常处理栈。如前所述，这是个单向链表的头节点。（你现在应该知道前面为什么要写`__this_gen->eh_list`了吧）

- `f` generator 所包装的函数指针。调用 generator 时，会在独立的栈空间中执行该函数。函数执行过程中可以多次进行 yield 操作。函数返回后，generator 将向外抛出`ERR_GENEND`异常标志执行结束。

你**需要**补全`gen.c`中的 yield、send 和 generator 函数的代码。`yield`将保存当前的上下文并恢复到 caller 的上下文，`send`将保存当前的上下文并恢复到目标 generator[14]的上下文。`yield`和`send`函数中可能还需要对涉及的 struct gen 结构体进行一些维护，请自行考虑。

`generator`接收一个函数指针 f 和一个参数 arg 作为参数。它将创建一个新的 struct gen 结构体，并对其进行一些初始化，保证通过 send 调用新建的 generator 后，程序会执行 f(arg)[15]，并在 f 返回之后抛出`ERR_GENEND`异常。如前所述，你可能需要仔细设置 struct gen 中的 ctx 字段，以使新建的 generator 能够在 recover 之后正确运行。

[13]通过 throw 形式

[14]send 的第一个参数

[15]中间可以经过一些跳板函数

你可能还**需要**修改你先前完成的`throw`代码，以在异常处理栈为空时，借助 struct gen 的 error 字段将异常抛出给 generator 的调用者。

如果一切顺利，完成本节代码后，你将通过 test5 和 test6。你也可以自己玩一玩

```
void counter(int n)
{
    for (int i = 0; i < n; i++)
        yield(i);
}
int main()
{
    struct gen* g = generator(counter, 233);
    try
    {
        while (1)
            printf("%d\n", next(g));
    } catch {}
    return 0;
}
```

至此，恭喜你已经顺利完成了整个 Kieray Lab！测试程序会输出一个彩蛋，你能认出他吗？

图 1: 彩蛋

5 提交

请将你的实验报告放在代码目录下，打包为**学号-lab3.tar**，提交到 elearning 上。

实验报告中可以包括下面内容

- **理论部分的答案**

- lab 的实现思路
- 对 lab 中有关内容的思考
- 对本门课程的意见和建议
- 其他任何你想写的内容~~，甚至可以放一只可爱猫猫~~

报告中不应有大段代码的复制。

实验报告不作为主要评分依据，正确通过全部测试即可得到大部分分数。

lab 的截止时间另行通知，请以 elearning 上显示的截止时间为准。