计算机系统基础

实验报告

Lab-3-AttackLab

姓名：马颢宸

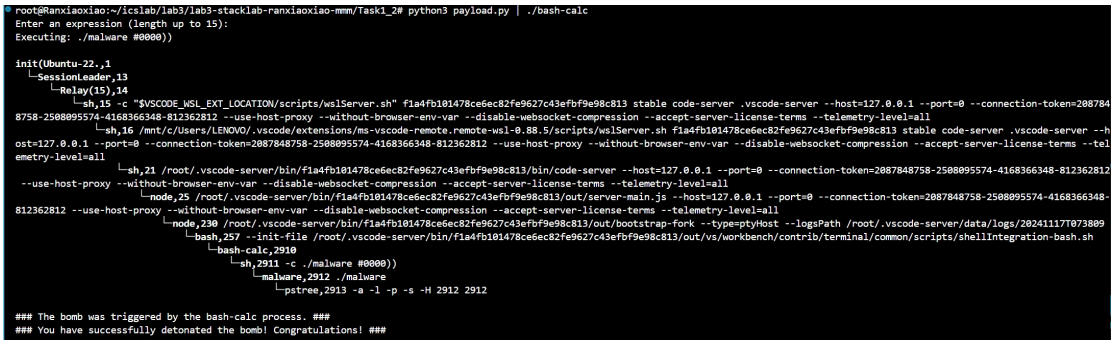
学号：23307110426

一、实验目的

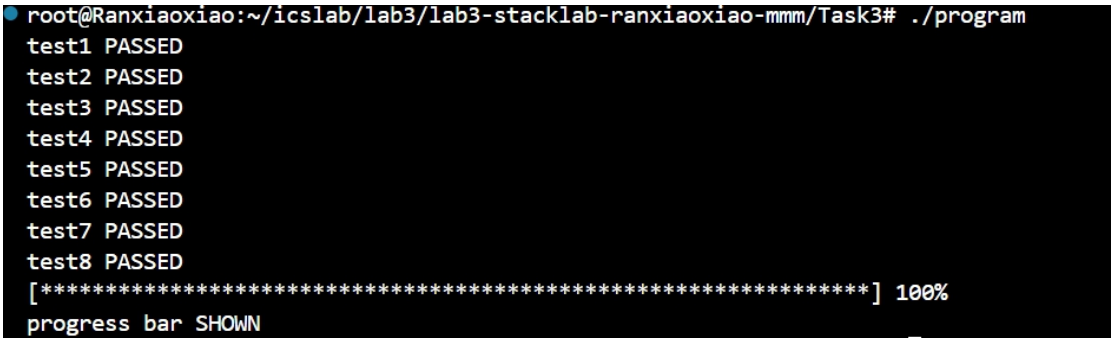
加深对栈帧的理解，并探索一些相关应用，丰富相关知识面。

二、实验结果

Task 1 调用重定向语句，成功完成任务



Task 3 任务完成截图



三、实验过程

Task 1

观察 bash-calc.c 源代码。发现需要劫持的关键函数 system 在 eval 函数中。主函数 main 先调用 input 函数，再调用 eval 函数。且两个函数共用一个参数。

观察 input 函数。input 参数为 cmd 字符串数组。在 input 函数中，先声明一个内存大小为 16 的 expression 字符串数组，再调用 gets 函数将命令行输入读取并存入数组。注意，此处 gets 函数在读取命令行输入时，会导致栈溢出问题。但 gets 函数所导致的栈溢出是在 expression 数组地址后发生的溢出，在 input 函数的栈帧内，若将 gets 函数作为栈溢出目标函数，则需更改其函数返回地址。但通过分析 input 的栈帧可知，无法通过 expression 数组的溢出修改 gets 函数的返回地址，故不考虑将 gets 作为目标函数。

input 函数通过 gets 函数和 strlen 函数判断命令行输入是否有效。若有效，则通过 strcpy 和 strncpy 函数将 wrapper_front[]（---> echo $((）、wrapper_back[]（--->))）和 expression 分别连接在 cmp 上，并返回 cmp。

故，欲通过栈溢出控制 bash-calc 程序执行恶意程序 malware，则一定需要利用 system 函数调用 ./malware，即 system（./malware）。通过上述分析，首先需要使输入命令行字符

串在不与 echo $((连接的情况下直接传入 system，否则会导致 system（echo $((./malware))），发生报错。

考虑第一种情况，即在 input 函数执行 strcpy 和 strncpy 函数前就使程序跳转至 system 函数。通过分析汇编代码，发现只有在 gets 函数处有修改栈中地址的机会。但根据上述分析，无法通过更改 gets 的返回地址调用 system 函数，故该情况 pass。

考虑第二种情况，即在分析 input 函数的汇编语言，并查看各寄存器在各阶段的值后发现，寄存器 rdi 在全程的过程中保存的对应地址共发生了四次变化，在 input 函数返回前的最后一次变化，保存地址为 0x404088，其对应的值为 expression 的第一个字符。所以，合理推断，input 函数返回前，rdi 寄存器中保存着指向 expression 的地址，即不与 echo $((连接的希望传入 system 的目标字符串。故接下来只需寻找将 rdi 寄存器传入 system 函数的路径即可。

因为 rdi 为函数调用的第一个参数，且分析 main 函数的汇编代码后发现，在 call input 和 call eval 语句之间还插入了一条 lea 0x2d2e(%rip),%rdi，即修改了第一个参数 rdi 的值。于是，合理推断此处将 input 中的 cmp 传入了 rdi 寄存器，并被 eval 函数作为参数调用。而 system 和 eval 调用同一个参数。故若能跳过该语句，则可将前述推断的 rdi 内的指向 expression 地址作为参数传给 eval 函数，即传给 system 函数。因为该语句对应地址为 input 函数的返回地址，所以只需利用 input 函数的栈溢出将该返回地址修改为 call eval 的返回地址即可跳过该语句。

阅读 input 函数的汇编代码可知，命令行输入存储在 0x20（$rbp）中，而 input 函数的返回地址在%rbp+0x8 处，故对应返回语句的偏移地址为 0x28。又因为 expression 数组内存大小为 16，即一定会读取输入命令行的前 16 个字符，故根据 md 文档提示，可构造前 16 个字符输入如下：./malware #00000。再利用 0 补足剩下的字符以达到返回地址处。根据 main 函数的汇编代码可知 call eval 的地址为 0x401352，故根据 md 文档提示的 python 文件构造 payload 即可。构造 payload 如下：

```
1  import sys
2  payload = b'./malware #'+b'0'*0x1D+b'\x52\x13\x40\x00\x00\x00\x00\x00'
3  sys.stdout.buffer.write(payload)
```

构造完毕后，在命令行输入 python3 payload.py | ./bash-calc 检验构造 payload 成功。

特别地，此处通过更改返回地址为 call eval 即实现通过中间函数调用 system 函数，存在 call system 语句，栈的对齐被破坏，不会导致 system 函数的内部崩溃。

Problem 1.1

在实际编程中，还有许多地方可能会存在栈溢出漏洞。

最常见的就是如本次实验的 gets 函数。在设置数组时，数组的内存大小不够使用，导致在运行时实际需要存放的量大于数组内存大小，数组访问越界，从而导致栈溢出。

在设置 while 循环语句时，若 while 的判断条件不得当 ，导致函数进入 while 的死循环，也会导致栈溢出。

在编写递归函数时，如果设置不得当，也会导致栈溢出。如忘记设置递归的结束条件时，函数将无限递归下去，而栈的大小有限，故会发生栈溢出。或在设置递归函数时，没有考虑到可能存在递归过深的情况，设置的递归次数过多，递归层次过深，也会导致栈溢出。

同理，设置多个函数相互调用时，若没有终止条件或为考虑嵌套过深的情况，也会导致保存返回地址和参数过多，从而导致栈溢出。

动态申请空间后没有释放或指针的非法访问也会导致栈溢出漏洞。如试图修改指针，使其指向一个非法地址时，也会导致栈溢出。

当然，也存在最简单的情形，即局部变量过大。如初始化一个数组时，数组内存大小设置过大，超过栈帧长度，也会导致栈溢出。

在简短的编程学习半年中，我曾在学习递归函数，编写斐波那契数列程序时遇见过栈溢出的清晰。根据斐波那契数列的特性，我编写代码如下：

```c
#include <stdio.h>

int f(int n) {
    if (n <= 1) return n;
    return f(n - 1) + f(n - 2);
}
```

由于调试时使用的数据都较小，函数可以正常进行。但在测试时，测试数据中存在一个非常大的数，导致函数递归次数过多，层次过深，从而产生栈溢出错误。在请教老师与查阅资料后，将该函数通过尾递归进行优化，即可避免栈溢出错误的发生。修改代码如下：

```c
#include <stdio.h>

long long f(int n, long long a, long long b) {
    if (n == 0) return a;
    if (n == 1) return b;
    return f(n - 1, b, a + b);
}
```

Problem 1.2

system 函数崩溃的原因是由于在使用 call 指令调用 system 函数时，栈上被压入了一个 8 字节的返回地址，从而破坏了栈对 0x10 的对齐。为了避免栈对齐被破坏的情况，system 函数假设自己在被调用时都是不向 0x10 对齐的。于是通过 push rbp 的指令调整栈，使之向 0x10 对齐。而若通过例如 jmp 等方式跳到 system 的起始地址，此时在 system 函数的入口处，因为未使用 call 指令，栈实际上的对齐的。但 system 函数默认自己未对齐，依旧 push rbp 去调整栈。矛盾的情形破坏了 system 函数的逻辑，从而导致程序崩溃。

而调用中间函数，并使用 jmp 等方式跳转到 system 函数，并且跳过函数开头的一条 push rbp 指令可以手动破坏栈的对齐，使得 system 函数可以正常执行。这是因为在调用中间函数时，函数入口默认栈未对齐。而此时跳过 push rbp 语句，栈仍旧处于未对齐状态。此时跳转至 system 语句，满足在 system 语句入口处栈未对齐的假设，system 函数通过调整栈使栈对齐，从而可以正常执行函数。所以，通过调用中间函数并跳过 push rbp 指令，会使得在 system 函数时栈未对齐，即近似实现了手动破坏栈的对齐。system 函数不会在内部崩溃，函数得以正常执行。

Task 2

首先利用 gcc -no-pie -o bash-calc-my bash-calc.c 语令重新编译 bash-calc-my，并对 bash-calc-my 进行重定向输入。即输入 python3 payload.py | ./bash-calc-my。命令行反馈如下：

```
root@Ranxiaoxiao:~/icslab/lab3/lab3-stacklab-ranxiaoxiao-mmm/task1_2# python3 payload.py | ./bash-calc-my
Enter an expression (length up to 15):
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

即检测到程序发生了栈溢出，并终止程序。故可知，bash-calc-my 程序较 bash-calc 程序新增了一个有关栈溢出的检测与防御机制。理论上，在 bash-calc-my 的源代码中应该比 bash-calc 的源代码多了几条检测栈溢出的语令，且可通过汇编代码观测出来。

利用 objdump -d bash-calc-my 语令打开 bash-calc-my 的汇编代码。再利用该语令打开 bash-calc 的汇编代码，并对两个汇编代码进行比较。明显发现 bash-calc-my 的代码较 bash-calc 代码多了__stack_chk_fail 函数，如：

```
00000000004010f0 <__stack_chk_fail@plt>:
  4010f0:       f3 0f 1e fa             endbr64
  4010f4:       f2 ff 25 35 2f 00 00    bnd jmp *0x2f35(%rip)        # 404030 <__stack_chk_fail@GLIBC_2.4>
  4010fb:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

```
40136d:       48 8b 55 f8             mov    -0x8(%rbp),%rdx
401371:       64 48 2b 14 25 28 00    sub    %fs:0x28,%rdx
401378:       00 00
40137a:       74 05                   je     401381 <input+0x102>
40137c:       e8 6f fd ff ff          call   4010f0 <__stack_chk_fail@plt> //
401381:       c9                      leave
401382:       c3                      ret
```

等处，都出现__stack_chk_fail 函数。同时，在 input 函数的开头还多出一段语令如下：

```
40128f:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
401296:       00 00
401298:       48 89 45 f8             mov    %rax,-0x8(%rbp)
40129c:       31 c0                   xor    %eax,%eax
```

仔细观察上述三段代码后发现，__stack_chk_fail 函数的触发条件是，当$rbp-0x8 中存储的值与$fs:0x28 中存储的值相减不等于 0（即二者不相等）时，会执行 call __stack_chk_fail 语令，从而触发栈溢出警告。而在 input 函数的开头，将$fs:0x28 中的值存入了$rbp-0x8 中。故，当且仅当$rbp-0x8 中的值被覆盖时，会触发__stack_chk_fail 函数。又因为$rbp-0x8 在栈中位于 input 函数的局部变量之前，在 input 函数的返回地址之后，故若$rbp-0x8 的值被覆盖，则 input 函数的返回地址极有可能被更改，即函数被恶意程序劫持。

使用 gdb 在 sub 处打下断点，并依次输入不同的输入进行调试。验证当输入存在溢出时，$rbp-0x8 中的值被覆盖更改，而不存在溢出时，值不变。同时发现，对于相同输入，$rbp-0x8 中存储的值并不相同。故对这种栈溢出防御机制的原理做出以下合理推测：

程序通过在$fs:0x28 处随机的存入一个值，并将该值也存入可能产生溢出的函数的高位栈帧中（在局部变量之前，紧邻函数的返回地址）。在函数执行结尾的 ret 语句前，通过 subq 指令，令$rbp-0x8 与$fs:0x28 中的值相减，若等于 0，则满足 je 条件，成功跳转到 ret 语句，函数继续执行。若不等于 0，则触发__stack_chk_fail 函数，程序被终止并发出栈溢出警告。

查阅相关资料后得知，这是 Stack Protector（栈保护机制）。当启用该机制后，在函数执行时，会向栈中插入一个 cookie 信息，函数返回时会验证 cookie 信息是否合法，如果不合法就终止程序。cookie 信息又称 canary 值，也就是上文中$fs:0x28 中存入的一个随机值。

Problem 2.1

不能。该机制实际是通过检测栈中 canary 值是否被覆盖更改来判断是否存在栈溢出。虽然 canary 值是随机产生的，但仍可存在值泄露的可能。即恶意程序可以在输入时刻意设置输入对应位置为 canary 值，从而使$rbp-0x8 被覆盖而值不变，欺骗检测机制。

也可通过外部程序更改$fs:0x28 中的值，使之与$rbp-0x8 中被覆盖后的值相等，从而欺骗检测机制。

故这种防御机制不能彻底的防御栈溢出漏洞。

Problem 3.1

内存-0x4(%rbp)处是 for 循环中判断循环条件变量 i 的值。

1217 位置的 mov 语句是为了取出-0x4(%rbp)处的值并存入寄存器 eax，从而再存入寄存器 edi。因为寄存器 edi 中保存的是函数调用的第一个参数，从而可将变量 i 作为 binSearch 函数的第一个参数传入，即实现语句 binSearch(i)。

Problem 3.2

首先，将变量保存在寄存器可以优化程序性能。因为访问寄存器所用时间比访问内存所用时间要快很多，特别当某些变量需要频繁访问时，使用寄存器存储比从内存中读取要节省很多时间。通过提高变量的访问速度，可以提高程序的执行效率。

其次，将变量保存在寄存器可以减少总线的占用。若从内存中读取数据，则需要 CPU 与内存进行频繁的交互。但若将变量保存在寄存器中，则减少了 CPU 对内存的访问，从而减少了总线的占用。

最后，将变量保存在寄存器中还可以减少空间的开销。栈上的变量一般需要进行入栈和出栈操作，而这些操作都涉及内存的分配与回收。若将变量保存在寄存器中，则避免了频繁的出入栈操作，从而可以减少空间的开销。

不能直接修改%rip 的值到 1221 而不做其他事情。若直接修改%rip 的值进行跳转，则存在寄存器或栈内的值并未更改，从而可能导致程序上下文失配。例如，在 call 指令的调用下，编译器会自动分配栈空间，保存函数的局部变量，在函数返回时会自动清理栈空间，使栈指针回到主函数的栈中。若直接修改%rip，则将跳过栈清理，导致栈指针%rsp 的指向不正确，从而影响主函数的后续执行。又因为 binSearch 函数利用寄存器保存局部变量，从而导致一些寄存器中存储的值被修改。若直接修改%rip 返回主函数，则这个寄存器的值未被恢复至主函数时的状态，从而影响主函数的后续执行。

由上述分析可知，还缺少如下步骤：

1、恢复各寄存器内的值

2、清理栈空间

3、恢复栈指针%rsp

Problem 3.3

除了修改%rip 以外，还需要复原寄存器和栈的状态（线程执行的状态）

Problem 3.4

假如变量 x 被存放在栈帧上，在 restore 操作后，x 的值为_1_；假如变量 x 被存放在寄存器上，在 restore 操作后，x 的值为_0_。

Problem 3.5

```
naive_func:
    endbr64
    mov (%rsp), %rax
```

```
mov %rax, (%rdi)
xor %eax, %eax
ret
```

Problem 3.6

固定格式 push rbp；mov rbp，rsp 是标准函数序言，用于建立栈帧。栈帧的主要作用是保存函数的局部变量、参数等。

因为在 naive_func 函数中，并不存在局部变量；函数的参数可以直接通过%rdi 传递，体量较小，无需存储在栈中访问；且该函数仅执行将返回地址保存到参数的操作，可通过%rsp 直接读取返回地址并存储在(%rip)中，不需要使用相关栈帧。

故可以不遵循标准函数序言的固定格式。

Task 3.1

首先进入 API 文件阅读有关__ctx_save 和__ctx_restore 函数的相关信息。__ctx_save 要求将上下文保存到 dst 引用的地址中，然后返回 0。__ctx_restore 要求将保存在 src 中的上下文恢复到寄存器中，并使 __ctx_save 返回值 ret_val。

对函数分析可知，save 函数仅含 dst 一个参数，存在%rdi 寄存器中，且根据函数的功能，可直接使用%rid，并且无需保存记录。save 函数需要保存上下文的状态，即调用前各寄存器的值、调用 save 的函数的栈底指针%rbp 和栈顶指针%rsp，以及 save 函数的返回地址（%rsp）。故根据上述分析，可利用 mov 指令逐个保存上述寄存器与栈相关值。特别地，在初调试时，为方便修改，暂时没有存储诸多寄存器的值（例如 rcx、rbx、r12 等）。但发现，即使不存储这些寄存器的值，仍然能成功 pass，考虑可能是测试函数并未将变量存放在这些寄存器中，所以巧合的不用保存。这也暗含一个常识，即大部分局部变量都放在栈里。但为保险起见，仍将这些寄存器都保存在 save 函数中。特别注意，在存储各变量时，对应地址需满足栈对齐的要求，即存储地址应按 8 字节对齐。

save 函数的第二个要求即是返回 0。因为%rax 中存储返回值，故可利用 xor 语句直接将%rax 内的值清零，即可满足条件。

对 restore 函数，恢复上下文操作即将 save 函数的 mov 指令反过来执行即可。注意%rbx 执行时的顺序。为使 restore 函数中的 ret_val 值返回为__ctx_save 的返回值，应首先将第二个参数%rsi（即存放 ret_val）存放在%rax 中，再将 save 函数的返回地址存放在（%rsp）。这样，在 restore 执行 ret 语句时，就会跳转到 save 函数的返回地址，此时的返回值%rax 中存放的就是 ret_val 而非 0，在函数外看来，就是 save 函数返回了 ret_val。

执行结果如下：

```
test1 PASSED
test2 PASSED
```

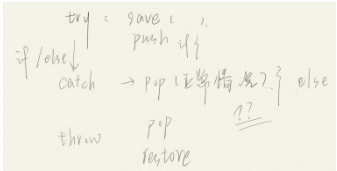
Task 3.2

```
void __err_stk_push(__ctx* ctx){
    assert(ctx != 0);
    __err_stk_node* new = (__err_stk_node*)malloc(sizeof(__err_stk_node));
    new->ctx = ctx;
    new->prev = __now_gen->__err_stk_head;
    __now_gen->__err_stk_head = new;

    // TODO
}

__ctx* __err_stk_pop(){
    assert(__now_gen->__err_stk_head != 0);
    __err_stk_node* top = __now_gen->__err_stk_head;
    __now_gen->__err_stk_head = top->prev;
    __ctx* popctx = top->ctx;
    free(top);
    return(popctx);

    // TODO
}
```

```
# define try \
    int __err_try __attribute__((cleanup(__err_cleanup))) = 0; \
    __ctx cur;\
    __err_stk_push(&cur);\
    __err_try = __ctx_save(&cur);\
    if(__err_try==0){

    // TODO

# define catch \
    }\
    else

    // TODO

# define throw(x) \
    {\
    __ctx* popctx;\
    popctx = __err_stk_pop();\
    __ctx_restore(popctx,x);\
    }
    // TODO
```

首先进入 API 文件阅读有关__err_stk_push 和__err_stk_pop 函数的相关信息。__err_stk_push 要求现在将 ctx 推入生成器的异常处理栈。__err_stk_pop 要求现在弹出生成器异常处理栈中的顶层元素。再分析__err_stk_node 结构体可知，链表结点有一个指向上一结点的指针和一个指向 ctx 的指针。由于 push 的参数为指向 ctx 的指针，故可知对 push 函数，只需使结点的 ctx 指针与参数相同，再将结点连入链表即可。对每一个新 push 入栈的 ctx，都应新建一个结点并更改其 ctx 指针与参数相同。特别地，由于结点含有指向上一结点的指针，故每个结点都应放在__now_gen->__err_stk_head 之后。但由于不存在尾指针，故应将链表头每次都改向最新加入链表的结点。即可认为，链表头在逻辑上是链表的尾结点。对 pop 函数，首先读取函数的链表头，即栈顶元素。根据链表的逻辑关系，将链表头改为指向原先链表头的上一元素，即栈中第二个元素，从而将链表头从链表中删去，实现 pop 目的。特别地，由于 pop 函数返回指向 ctx 的指针。故建立 ctx 指针型变量并使之等于读取元素的 ctx 值。通过 free 指令释放老链表头。最后返回指向 ctx 的指针。

对三个宏定义，首先根据 readme 文档判断 try、catch 和 throw 中需要进行什么操作，即：（思考草图）



对正常完成的 try 操作，应完整走完 if 内语句，故此处考虑将 pop 函数放在 catch 内，但在 catch 的 else 之前，即可保证能在做完 try 操作后的语句再 pop，同时 pop 仍属于 if 语句内，不会被 throw 影响。（若放在 else 之后，则在 throw 做完 pop 指令后，进入 catch 语句将再做一次 pop 指令，不符合逻辑）

根据上述逻辑，编写宏定义如下：

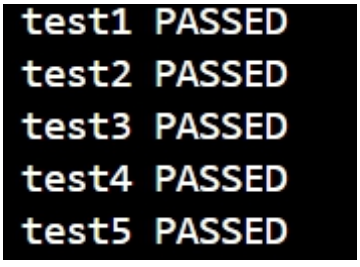
但运行后发现，虽然 test3、test5 都能成功通过，但 test4 一直无法输出。使用 gdb 调试后发现，程序终止在第 113 行，且栈顶存放函数__pthread_kill_implementation。调出函数__pthread_kill_implementation 的汇编代码后发现，程序终止在 0x7ffff7e229fc <pthread_kill+300>。查阅相关资料后得知，pthread_kill 函数是 linux 中有关线程的信号函数。可判断该线程是否还“活着”。又根据 readme 文档中提示，cleanup 属性会在对应变量结束生命周期时执行对应函数。故猜测此处的生命周期结束即对应线程“死去”，pthread_kill 函数应是 cleanup 属性的判断函数。

回看 try 操作的宏定义发现，cleaup 对应变量为__err_try，即当__err_try 对应程序线程结束时，cleanup 会被启用、pthread_kil 被调用、cleanup 被启用，test4 崩溃。同时，发现 cleanup 会调用函数__err_cleanup，而此时该函数程序仍未定义。

故，考虑补全__err_cleanup 函数。因为当__err_cleanup 函数被调用时，对应__err_try 生命周期结束。而__err_try 定义在 try 操作内，即应对应 try 操作完成。在 readme 中强调，当 try 正常结束时，也需要执行 pop 操作。而在之前的代码中将这个 pop 放在了 catch 操作中。但这种做法显然失败了。于是考虑将 pop 操作放在__err_cleanup 函数中。try 操作正常完成的标志是__err_try=0，即未执行 throw 操作->未执行 restore 函数->未改变 save 函数的返回值->__err_try=0。又因为__err_cleanup 函数的参数是指向__err_try 的指针，故可以直接利用该指针在函数中判断__err_try 的值即可，若__err_try=0，则执行 pop 函数。同时，删去 catch 宏定义中有关 pop 的语句。__err_cleanup 函数如下：

```
void __err_cleanup(const int* error){
    if(*error == 0){
        __ctx* popctx;
        popctx = __err_stk_pop();
    }
    // TODO
}
```

运行后发现，test3、4、5 均成功通过。



Task 3.3

首先进入 API 文件阅读有关 struct __generator 函数的相关信息。发现该结构体具有如下几个重要变量：生成器函数*f、整数型变量 data、上下文数组 ctx、栈 stack、调用者生成器*caller、异常处理栈链表头*__err_stk_head。目标函数 generator 为结构体的构造函数，故先利用 malloc 函数创建一个结构体指针变量；利用 memset 函数将生成器中各值先清零，以防止出现奇怪的数据；将 generator 的传入参数 f 赋值给 gen->f；利用 malloc 函数分配栈空间；将 caller 指向主函数，以方便后续恢复操作；因为此时还未执行 try，故异常处理栈链表头置为 0。且需要手动在内存中配置一段初始上下文，并存入包括栈指针、栈基址、返回函数、以及函数参数。对于栈指针，由于调用时有栈对齐要求，故首先需要计算栈的起始空间使其满足 16 字节对齐，否则后续就会出现 segmentation 报错。手动创建一个 long long 型的指针变量并使其等于 gen->ctx（这里需要强制类型转换）。根据 save 函数的存储顺序，依次将对于参数存放在数组对应位置。特别地，返回函数地址存放跳板函数 gen_tram。跳板函数主要用在启动生成器，即在当前生成器下跳转到目标生成器函数并执行，执行结束后返回跳板函数并 throw。而参数 arg 为生成器函数参数，需要先传入跳板函数，再传入生成器函数。故将 arg 设置在 rdi 寄存器位置，既可在 restore 时作为第一参数传入。generator 构造函数和跳板函数代码如下：

```
__generator* generator(void (*f)(int), int arg) {
    __generator* gen = (__generator*)malloc(sizeof(__generator));
    memset(gen,0,sizeof(__generator));
    gen->f = f;
    gen->stack = malloc(10000);
    gen->caller = __now_gen;
    gen->__err_stk_head = NULL;

    //计算栈起始空间
    char* stack_top = (char*)gen->stack + 10000;
    while((unsigned long)stack_top % 16 != 0){
        stack_top--;
    }
    long long* handctx = (long long*)gen->ctx;
    *handctx = (long long)gen_tram;
    *(handctx+2) = (long long)stack_top;
    *(handctx+1) = (long long)stack_top;
    *(handctx+10) = (long long)arg;
    return gen;
    // TODO: construct a new generator
}
```

```
void gen_tram(int arg){
    __generator* gen = __now_gen;
    gen->f(arg);
    throw(ERR_GENEND);
    assert(0);
}
```

对于 send 函数，因为 send 操作是从主生成器恢复到创建生成器的上下文，故首先将主生成器赋值为创建生成器的 caller，然后 save 主生成器的上下文并判断此时处于保存状态还是恢复状态，若为保存状态（=0），则将主生成器换为创建生成器并 restore 创建生成器的上下文；若为恢复状态，则返回创建生成器的 data 值，即 yield 向 send 传送的值。

对于 yield 函数，大致框架和 send 函数相同，因为二者操作基本对称。区别是，yield 函数需要手动创建一个 generator 型指针变量以保存主生成器的 caller。后续操作相似。Yield 函数返回主生成器的 data 值，即 send 向 yield 传送的值。send 和 yield 的代码如下：

```c
int send(__generator* gen, int value) {
    if (gen == 0) throw(ERR_GENNIL);
    gen->data = value;
    gen->caller = __now_gen;
    if(__ctx_save(&(__now_gen->ctx)) == 0){
        __now_gen = gen;
        __ctx_restore(&(gen->ctx),1);
    }
    // TODO

    return gen->data;
}

int yield(int value) {
    if (__now_gen->caller == 0) throw(ERR_GENNIL);
    __now_gen->data = value;
    __generator* call_gen = __now_gen->caller;
    call_gen->data = value;
    if(__ctx_save(&(__now_gen->ctx)) == 0){
        __now_gen = call_gen;
        __ctx_restore(&(call_gen->ctx),1);
    }
    // TODO

    return __now_gen->data;
}
```

根据 readme 文档的提示，此时还需修改 throw 宏。因为存在一种可能，当生成器的异常处理栈为空，但是有需要 throw 时，throw 应该返回离他最近的一个 try 宏处。即，当函数在 throw 前未进行 try 操作时，需要利用 throw 操作返回一个正常值，否则就会因为 pop 了一个空栈而报错。但因为可能出现连续多个 throw 函数的当前生成器中都没有 try 函数，故对 throw 应添加一个递归定义。又因为宏定义无法递归，故增加跳板函数 gen_throw，仅含 throw 宏，以实现在宏定义中递归调用。故对宏定义，先判断异常处理栈是否为空，再将当前栈和目标栈调换，最后递归调用直至栈不为空，成功 pop 并 restore。throw 宏和跳板函数代码如下：

```c
# define throw(x) \
    if(__now_gen->__err_stk_head == NULL){\
        __now_gen = __now_gen->caller;\
        gen_throw(x);\
    }\
    else\
    {\
        __ctx* popctx;\
        popctx = __err_stk_pop();\
        __ctx_restore(popctx,x);\
    }\
    // TODO
```

```c
void gen_throw(int x){
    throw(x);
    assert(0);
}
```

运行后发现，test6、7、8 均正常通过。

```
test6 PASSED
test7 PASSED
test8 PASSED
```

Task 3.4

这个 Task 整体难度比前面的 Task 低很多，因为所有的线程相关函数都已搭配好，只需编写运行界面的函数即可。由于是输出进度条，故根据 progress 每个协程中不同的值，利用 for 循环一句逐个打印处想要的字符串即可。十分简易。代码如下：

```
void progress_bar(int init){
  int progress = init;
  while(1){
    printf("\r[");
    for(int i = 0; i < 64;i++){
      if(i <= progress){
        printf("*");
      }
      else{
        printf(" ");
      }
    }
    printf("] %d%%",((progress+1)*100)/64);
    // TODO: Implement an animation of progress bar
    fflush(stdout);
    progress = yield(progress);
    if (progress == 64) break;
  }
}
```

运行结果如下：

```
test1 PASSED
test2 PASSED
test3 PASSED
test4 PASSED
test5 PASSED
test6 PASSED
test7 PASSED
test8 PASSED
[************************************************************] 100%
progress bar SHOWN
```

四、参考资料

https://blog.csdn.net/qq_64318258/article/details/126236587?fromshare=blogdetail&sharetype=blogdetail&sharerId=126236587&sharerefer=PC&sharesource=2402_84096879&sharefrom=from_link

https://blog.csdn.net/qq_41573572/article/details/136891830?fromshare=blogdetail&sharetype=blogdetail&sharerId=136891830&sharerefer=PC&sharesource=2402_84096879&sharefrom=from_link

https://atfwus.blog.csdn.net/article/details/104552315?fromshare=blogdetail&sharetype=blogdetail&sharerId=104552315&sharerefer=PC&sharesource=2402_84096879&sharefrom=from_link

https://geekcode.blog.csdn.net/article/details/135754690?fromshare=blogdetail&sharetype=blogdetail&sharerId=135754690&sharerefer=PC&sharesource=2402_84096879&sharefrom=from_link

https://blog.csdn.net/zang141588761/article/details/103984226?fromshare=blogdetail&sharetype=blogdetail&sharerId=103984226&sharerefer=PC&sharesource=2402_84096879&sharefrom=from_link

https://blog.csdn.net/weixin_37357702/article/details/121622518?fromshare=blogdetail&sharetype=blogdetail&sharerId=121622518&sharerefer=PC&sharesource=2402_84096879&sharefrom=from_link

https://blog.csdn.net/EleganceJiaBao/article/details/141252278?fromshare=blogdetail&sharetype=blogdetail&sharerId=141252278&sharerefer=PC&sharesource=2402_84096879&sharefrom=from_link

https://gxdxyl.blog.csdn.net/article/details/136723966?fromshare=blogdetail&sharetype=blogdetail&sharerId=136723966&sharerefer=PC&sharesource=2402_84096879&sharefrom=from_link

https://lkmao.blog.csdn.net/article/details/138124713?fromshare=blogdetail&sharetype=blogdetail&sharerId=138124713&sharerefer=PC&sharesource=2402_84096879&sharefrom=from_link

https://blog.csdn.net/weixin_55772194/article/details/123153688?fromshare=blogdetail&sharetype=blogdetail&sharerId=123153688&sharerefer=PC&sharesource=2402_84096879&sharefrom=from_link

https://blog.csdn.net/GoodLinGL/article/details/114602721?fromshare=blogdetail&sharetype=blogdetail&sharerId=114602721&sharerefer=PC&sharesource=2402_84096879&sharefrom=from_link

五、对实验的感受

好难好难好难 T^T 感觉有很多需要细节细化的代码，虽然 readme 讲得很细致了，但对于 0 上手的我们来说，还是有很多地方比较抽象。这次真的学到了特别多新的东西，查了很多资料（感觉每做一道题就要看很多知识讲解才能理解 lab 想要我做什么。）总之，这次 lab 很有挑战性，只是感觉趣味性没有 bomblab 那么强，卡在同一个点 debug 了一整天后确实比较暴躁。还好最后都做出来了^ ^。

六、对助教的建议

如果下次 lab 难度很大，可以不要出在期中季吗 kuku。
希望这是最难的一个 lab 了。
不要再上强度了 www。