ICS Lab3
理论部分

Fudan-ICS

2022 年 10 月 13 日

本文档是复旦大学计算机科学技术学院 2022 年秋季《计算机系统基础》课程实验三的理论部分指导文档。**理论部分答案计入实验总分**，请认真思考、独立完成本文档中提出的所有问题，是否正确理解理论部分所有问题是能否顺利完成实验部分任务的关键。

本文档中，所有要求回答的问题均以 Problem 形式明确标注，其他问题仅希望大家思考，不要求回答。**如无特殊说明，均要求使用 x86-64 指令集**。在书写汇编时，你可以自由选用 Intel Syntax 或 AT&T Syntax，但不得两者混用。

1 数蘑菇的小喵喵

虎鲸很喜欢上金老师的课，因为金老师上课寓教于乐，和同学们谈笑风生，比某些老师[1]不知道高到哪里去了。这么亦可赛艇的课堂，想必大家一定都认真听课了吧。那么下面的显然是送分题。

Problem 1 请写出x86-64调用约定中所有的 *callee-saved GPR*[2]。

Problem 2 请用不超过 5 条汇编指令[3]实现函数naive_func。该函数的功能为：将函数的返回地址保存到第一个参数所指定的内存地址，然后返回 0。可参考伪代码:`naive_func(void **p): *p = (return addr); return 0;`

Problem 3 也许你在课上学过函数开头两条指令的固定格式[4]。但是前面所写的naive_func显然不遵循这样的格式，为什么这是可以的呢？

[1] L■■■■

[2] General Purpose Registers

[3] 不计伪指令。如果可以，最好在函数开头加上 endbr64，指令数量应该是够的。

[4] 指 push rbp; mov rbp, rsp

小喵喵也上了金老师的课[5]。在听到调用约定的知识后，正在数蘑菇[6][7]的小喵喵想起了忒修斯之船的故事。

> 忒修斯之船最早出自普鲁塔克的记载，它描述的是一艘可以在海上航行几百年的船，归功于不间断的维修和替换部件。只要一块木板腐烂了，它就会被替换掉，以此类推，直到所有的功能部件都不是最开始的那些了。问题是，最终产生的这艘船是否还是原来的那艘特修斯之船，还是一艘完全不同的船？如果不是原来的船，那么在什么时候它不再是原来的船了？

为方便说明问题，小喵喵写了下面的伪代码

```
func A:
    t = call B
    print (t)
    return t * 2 - 1
func B:
    return 233
```

我们知道，在函数 A 调用函数 B 时，函数 A 会将返回地址压栈，在函数 B 返回时，它会将返回地址出栈并跳转到返回地址。但是，在函数 B 返回之后，函数 A 能正确地继续执行，并不只是因为程序跳转到了设定的返回地址处。假如我们在程序运行之初，通过某种外部手段[8]将 PC 直接改到函数 A 调用函数 B 时设定的返回地址处，它显然是不能如同函数 B 返回了一样正确地往下执行的。

那么，同样是 PC 从函数 A 的外部跳转到了函数 A 内，且跳转到的是同一个位置，甚至给出的 rax（在函数 A 中将被视作返回值）也是同一个，函数 A 调用函数 B 后函数 B 的返回和直接借助外部手段的强行跳转，他们的区别究竟在哪呢？

Problem 4 *除了将 PC 设置到返回地址以外，还需要设置哪些内容，才可以构造出一个如同函数 A 调用函数 B 后函数 B 的返回一样的环境，使函数 A 可以正确地继续执行？*

假定函数 A 和函数 B 只使用寄存器和栈内存，不使用堆内存、数据内存甚至操作系统提供的外部资源。

有一个最简单的思路是，提前运行一遍程序，将函数 B 即将返回时程序的所有内存和寄存器值都存下来。在再次运行程序时，直接用保存的内存

[5] 他好像真的上过

[6] https://www.bilibili.com/video/BV1mV4y1W7Ze/

[7] 不要在意为什么小喵喵在课上数蘑菇——就像有的同学可能也没在听课一样

[8] 比如，在 gdb 中执行 set $pc=0x12345678，甚至可以把 rax（返回值）也设成 233

和寄存器值覆盖，这样肯定是可以的，但显然意义不大。我们希望你能尽可能少保存或设置内容。

前面所说的那个决定了一个程序执行状态的“环境”，我们称为**上下文**。参考 Wikipedia，我们给出这个概念更严格一些的定义

> 任务的上下文（context）是一个任务所必不可少的一组数据。这些数据完全描述了这个任务的执行状态，通过存储这些数据，我们可以将任务暂停并在另一个地方恢复正常执行，也可以在不影响执行的情况下复制任务。

2 旅行青蛙

金蟾乐园的池塘里，有一只三只耳朵的小青蛙[9]，他最大的爱好就是出去旅行。某一天，小青蛙在外旅行时，遇见了一只狐狸。狐狸告诉他，在那云雾缭绕的山峰上，有着一棵神秘的树，它的叶子都是黑色，枝条则是红黑相间，非常好看。狡猾的狐狸欺骗青蛙：摘下一截黑色的枝条，你就可以实现一个愿望。小青蛙听信了狐狸的谎言，摘下枝条之后，树却轰然倒塌，将小青蛙埋在了下面……幸运的是，小青蛙会时间魔法[10]。在摘下枝条之前，小青蛙就将那个时刻的自己和红黑树[11]都复制了下来，在树即将倒塌的时候，小青蛙利用原先的复制将自身和红黑树恢复到了摘下枝条之前，仿佛时光倒流了一样。这一次，小青蛙摘下了红色的枝条，成功实现了自己的愿望……

[9] 他的名字叫乔治·魄罗蛙

[10] ~~青蛙会时间魔法不是很合理吗 ovo~~

[11] 指山峰上的那棵树。因为它红黑相间，我们叫它红黑树。

我们可以用伪代码来表达上面的故事

```
try
    摘黑色枝条
catch
    摘红色枝条
```

主流编程语言中都有 try-catch[12]，想必不用我们再向大家介绍。

Problem 5 请简述一种常见编程语言（如 *C++*、*Java*、*Python* 等）的 *try-catch* 语法规则。

try-catch 的实现原理非常多样。一般情况下，实现 try-catch 需要编译器的支持。但上面的故事或许能给我们一些启发。

Problem 6 请参考第一节所讲的内容，设计一种不依赖于编译器支持，通过“时光倒流”实现 *try-catch* 的办法。

不需要给出实现细节。

“时光倒流”的想法可追溯到 OSDI'08[13]，常见调试工具都已实现了此类功能，如 gdb 的 record&replay（rr）[14]、windbg 的 time travel（微软起名还是那么的唬人）。

3 Python 精通者

Python 可以说是世界上最简单的编程语言之一，想必同学们在大一的时候就已经很精通了[15]。下面，我们先帮大家复习一下 Python 中的 generator 机制[16]。当一个函数中含有 yield 时，他就成为一个 generator。每次执行 yield，函数都会临时返回一个值，然后可以通过 next 或 send 让函数在 yield 的下一句继续执行。

可参考下面代码

```
def gen(n):
    for i in range(n):
        yield i
```

[12] 指编程语言中与语言内部的 throw、raise 等关键字互动的 try-catch。有些编程语言还会将 try-catch 与操作系统异常处理挂钩，但这不是必要的。

[13] Z.Guo,X.Wang,J.Tang,X.Liu,Z.Xu,M.Wu,M.F. Kaashoek, and Z. Zhang. R2: an application-level kernel for record and replay. OSDI' 08, 2008.

[14] ~~lrx 经典翻车现场~~

[15] ~~不会吧不会吧，不会有人还没被电导课荼毒过吧~~

[16] 如果你真的不会，可以自己百度，也可以参考这篇也许写的不错的博客https://www.liaoxuefeng.com/wiki/1016959663602400/1017318207388128

```
g = gen(10)
while True:
    print(next(g))
```

Problem 7 上面代码会输出什么呢？[17]

希望上面那个简单的例子能够帮助大家回忆起 Python 的一些基础知识。generator 是一种非常棒的设计，它可以轻易地将普通函数改造为迭代器。lrx 是 Python 精通者，他希望 C 语言中也有这样的好功能，你能帮帮他吗？

Problem 8 请参考第一节所讲的内容，设计一种不依赖于编译器支持，在 *C* 语言中实现 *Python* 的 *generator* 机制的办法。

不需要给出实现细节。

Python 中的 generator 就像一个普通函数一样，会向外抛出未处理的异常，可参考

```
def gen(n):
    for i in range(n)
        try:
            yield do_something(i)
        except KeyboardInterrupt:
            raise KeyboardInterrupt
        except Exception:
            pass
```

当发生 KeyboardInterrupt 时，gen 的调用者[18]将收到抛出的该异常。你能将前面设计的 generator 和 try-catch 结合，让你的 generator 也能够向外抛出未处理的异常吗？

Problem 9 请将前面设计的 *generator* 和 *try-catch* 机制结合。如何让 *generator* 中未处理的异常被抛出给调用者呢？

Python 中的 generator 是协程（coroutine）的一种表现形式。如果你对此有兴趣，可以自行阅读一些扩展资料，ICS 课程中对此不作要求。

[17] 你可以自己打开 Python 试一试

[18] 即 next(gen) 的调用者