汇编：作业3

刘泓尊 2018011446 计84 liu-hz18@mails.tsinghua.edu.cn

1.SET & GET

1.1 `SET`过程的返回地址是什么，其返回值是多少?

`SET`的返回地址是调用`GET`函数(`call GET`)的**下一个**指令，也就是`GET`的返回地址

返回值存在`%eax`中，即内存块中`48(%eax)`的值，为1。

1.2（A）指令执行后，`%eax`中存放的是"存储当前处理器以及栈信息的内存块"（下简称"调用环境内存块"）的地址。

实际上，此时`movl 4(%esp), %eax`就是将保存`current_context`的地址存入`%eax`

（B）指令执行后，`%ecx`中存放的是`GET`函数的返回地址。

（C）指令的作用是 将在栈中指向"调用环境内存块"的指针(也就是`%eax`的值)的地址写入`%ecx`.(最终将被下一条指令"`mov %ecx 72(%eax)`"存进"调用环境内存块")。

一句话来讲，此时`(%eax) == %ecx`

（E）指令的作用是 将之前保存在"调用环境内存块"的`GET`函数的返回地址压栈。

（D）指令：`movl 72(%eax), %esp`

下面详细分析一下"调用环境内存块"的字段与存储信息, `M[]`表示内存:

```
//调用GET之后，内存块存储的信息
 M[%eax+72]    |    %esp+4    //注意此处不是 M[%esp+4]
 M[%eax+68]    |    ---
 M[%eax+64]    |    ---
 M[%eax+60]    |    M[%esp]   //即 GET函数 的返回地址
 M[%eax+56]    |    ---
 M[%eax+52]    |    ---
 M[%eax+48]    |    $1        //将来由SET函数返回
 M[%eax+44]    |    %ecx      //以下寄存器信息由SET恢复
 M[%eax+40]    |    %edx
 M[%eax+36]    |    ---
 M[%eax+32]    |    %ebx
 M[%eax+28]    |    %ebp
 M[%eax+24]    |    %esi
 M[%eax+20]    |    %edi
 ...
 M[%eax]       |    ---       //%eax 为指向该内存块的地址
```

当调用`SET`之后，在`movl 48(%eax), %eax`执行之前，栈的情况为(记`GET`函数中`%esp`的值为`%esp_old`)

```
%esp+8  |    %eax          //%eax 为指向内存块的地址
%esp+4  |    %esp_old+4
%esp    |    M[%esp_old]   //即 GET函数 的返回地址
```

执行`movl 48(%eax), %eax`后，`%eax`将为1，作为`SET`函数的返回值。

执行`ret`之后，`SET`函数将返回到`GET`函数的返回地址，实现**非本地跳转**，而不需要解开调用栈。

可以看到，非本地跳转允许了控制流从一个深层嵌套的函数中返回，为异常处理和协程切换提供了可能。实际上，`GET`和`SET`过程与C标准库中的宏`setjmp()`和`longjmp()`等价。上述调用`GET`和`SET`的过程可以等价于如下C代码：

```c
#include <setjmp.h> /* jmp_buf, setjmp, longjmp */
int main() {
  jmp_buf env;
  int val;
  val = setjmp (env); //A: return 0 first time, return 1 second time
  if (val) {
    fprintf (stderr,"Error %d happened",val);
    exit (val);
  }
  longjmp (env, 1);   // signaling an error, jump to A
  return 0;
}
```

2.协程

2.1 工作原理

`movq current_ctx, %rsi`将存储当前上下文的地址存入寄存器`%rsi`。然后将当前寄存器堆信息(`%rsp, %rbx, %rbp, %r12, %r13, %r14, %r15`)依次存入内存块`M[%rsi]-M[%rsi+48]`的内存中（保存现场）。实际上这些寄存器便是`callee saved register`和`栈指针register`。

之后将`%rdi`的值（指向新任务的上下文内存块）存入`current_ctx`，作为当前新的上下文，便于将来保存新任务的上下文。

随后加载新的寄存器信息（这些信息存储在`%rdi`指向的内存块中）。将`%rsp, %rbx, %rbp, %r12, %r13, %r14, %r15`依次装载入寄存器（恢复现场），实现寄存器的上下文切换。

2.2 为何`save/load`的通用寄存器个数这么少?

`save/load`的寄存器只需要保存**被调用者保存寄存器**(`%rbx, %rbp, %r12, %r13, %r14, %r15`)和**栈顶指针寄存器**(`%rsp`)。

在不存在协程切换的场景下，**被调用者保存寄存器**(callee saved)需要由**被调用者**(callee)负责，这样保证了其他函数能安全地使用这些寄存器。**但是**，因为协程调度是跨越栈的，不存在`caller-callee`的区分，所以**当从任务A切换到任务B后，B不会为A恢复**`callee saved register`，所以这些寄存器要保存到A的上下文中。而对于调用者保存寄存器(`caller saved`)，因为调用`switch`进行切换，所以A会自动保存`caller-saved-register`，不需要再次保存。因此，通用寄存器中只需要保存**callee-saved-registers**。

当然，**栈指针寄存器**`%rsp`的保存是必需的，它保证了函数的正确返回与临时变量的正确存储。

实际上，上述上下文信息可以保存在如下结构中：

```
typedef struct {
    /*
     * buffer[0]: rsp
     * buffer[1]: rbx
     * buffer[2]: rbp
     * buffer[3]: r12
     * buffer[4]: r13
     * buffer[5]: r14
     * buffer[6]: r15
     * buffer[7]: Program Counter
     */
    long buffer[8];
} ctx_buf_t;
```

补充：条件码`Condition Code`不需要保存：

因为协程切换是在**用户态**下进行，对于条件控制指令，用户态是不能强行中断的。也就是`Condition Code`的set and test操作对于用户态而言是**原子**的。所以不需要保存条件码。

3.`struct`类型是如何实现的

`call return_struct`前，`caller`为存储`struct`开辟了32字节的栈空间(`subq $32 %rsp`).

之后caller将当前栈指针`%rsp`传入`%rdi`，作为参数传入`return_struct`。

在`return_struct`返回后，栈的layout为

```
|   ret后   |    内容
| %rsp+16   |  %esi
| %rsp+12   |  %esi
| %rsp+8    |  2*%esi
| %rsp+4    |  %esi
| %rsp      |  %esi
```

返回值放在`%rax`中，也就是`%rdi`的值。

综上，`struct`通过指针返回，caller先在内存中开辟空间，然后函数返回指向这个空间的指针（首地址）。

4.C函数是如何传入`struct`类型参数的

4.1 `gcc -Og`

`call input_struct`时栈的layout：

```
%rsp+48  |  ----
%rsp+44  |  ----
%rsp+40  |  ----
%rsp+36  |  %eax     # ddd
%rsp+32  |  2*%eax   # coo
%rsp+28  |  %eax     # bye
%rsp+24  |  %eax     # age
%rsp+20  |  ----
%rsp+16  |  %eax     # eee ->24(%rsp) in input_struct
%rsp+12  |  ----
%rsp+8   |  2*%eax   # coo
%rsp+4   |  ----
%rsp     |  %eax     # age ->8(%rsp) in input_struct
```

在进入`input_struct`后，`%rsp`下移8字节，此时`8(%rsp)`保存的为`age`，`24(%rsp)`保存的为`eee`.`input_struct`计算结果存入`%eax`中返回。

综上，在`-Og`开关下，`struct`作为参数传递时，`caller`**在栈中开辟空间**存储`struct`的值（可能带来多余的空间），`callee`只需要取栈上相应地址的值即可。

4.2 `gcc -O1/2`

编译器认识到了`input_struct`的返回值仅仅是`3i`，因此并没有实际调用`input_struct`函数，而是直接将`i`传入寄存器`%eax`，并将其乘3，作为`function2`的返回值返回。

但同时保留了`input_struct`的汇编代码，并且认为`eee`和`age`字段的内存分别存储在`24(%rsp)`和`8(%rsp)`中，其余字段并没有产生对应的赋值指令。这极大提升了性能。

保留`input_struct`的汇编代码是为了保证**其他文件**的调用。

4.3 函数前加上`static`，`gcc -O1/2`。编译器的优化

并没有生成`input_struct`的汇编代码，编译器将其`inline`掉了。因为`static`函数只是在本文件中使用，既然整个过程可以被充分优化为`leal (%rax, %rax, 2), %eax`，那么便没有必要生成`input_struct`的汇编代码。

补充：

要防止函数被`inline`，需要在函数声明前加上`__attribute_used__`