

# 并行与分布式计算

## 第三次作业

姓名：张洪宾

班级：18 级计科超算

学号：18340208

# 1 问题描述

利用 LLVM (C、C++) 或者 Soot (Java) 等工具检测多线程程序中潜在的数据竞争以及是否存在不可重入函数，给出案例程序并提交分析报告。

## 2 实验环境

### 2.1 软件

- macOS Catalina with Apple clang version 11.0.3
- Ubuntu Linux 18.04 with clang version 6.0.0-1ubuntu2

### 2.2 硬件

所用机器型号是 MacBook Pro2015

- 2.5 GHz 四核 Intel Core i7
- 16 GB 1600 MHz DDR3

## 3 解决方案

此次作业主要有两部分组成，一部分是用 clang 的 ThreadSanitizer 工具对多线程程序中潜在的 data races 进行检测，一部分是通过分析 llvm 生成的中间代码 IR，判断函数是否为可重入函数。

### 3.1 使用 ThreadSanitizer 工具检测多线程程序中潜在的 data races

多线程编程多种多样，可以选择的有 pthread、MPI、openMP 等方式，由于之后的课程会涉及到 openMP 和 MPI 编程，而我最近又刚好在学习 pthread，所以这一次我使用 pthread 编程。

为了设计出合适的数据冒险并进行分析，并适当地锻炼自己的编程能力，我找到了一个比较适合例子：用麦克劳林展开式估算  $e$  的值。由麦克劳林展开式很容易得到：

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

由于级数的各项不存在数据相关，比较符合并行编程的条件，所以用这个例子可以比较全面地说明问题，并学会使用 ThreadSanitizer 工具来检测我们的程序是否有数据竞争的问题。

## 3.2 分析 LLVM 中间代码 IR 来判断是否有不可重入函数

### 3.2.1 关于 LLVM

由于还没有学习编译原理，在最开始我不太理解为什么老师比较推崇 LLVM 而不是 gcc，直到我读到了一篇文章：[The Architecture of Open Source Applications: LLVM](#)。从这篇文章我才理解了 LLVM 在开发上的优势。

LLVM 与 GCC 在三段式架构上并没有本质区别，但是由于 gcc 在最开始开发的时候并没有设计成 Reuse 的方式，正如文中提到：

The hardest problems to fix, though, are the inherent architectural problems that stem from its early design and age. Specifically, GCC suffers from layering problems and leaky abstractions: the back end walks front-end ASTs to generate debug info, the front ends generate back-end data structures, and the entire compiler depends on global data structures set up by the command line interface.

所以开发者很难重用 gcc 的库。而 LLVM 在一开始设计的时候就在比较高的角度，制定了 LLVM IR 这一中间代码表示语言。LLVM IR 充分考虑了各种应用场景，例如在 IDE 中调用 LLVM 进行实时的代码语法检查，对静态语言、动态语言的编译、优化等。所以我们如果开发一个编译器只需要实现前端即可，这也是 clang 的做法。

而我也逐渐理解老师在这门课程开始的时候提出让我们利用 LLVM 开发一门语言的用意，也找到了完成这个终极目标的途径了。

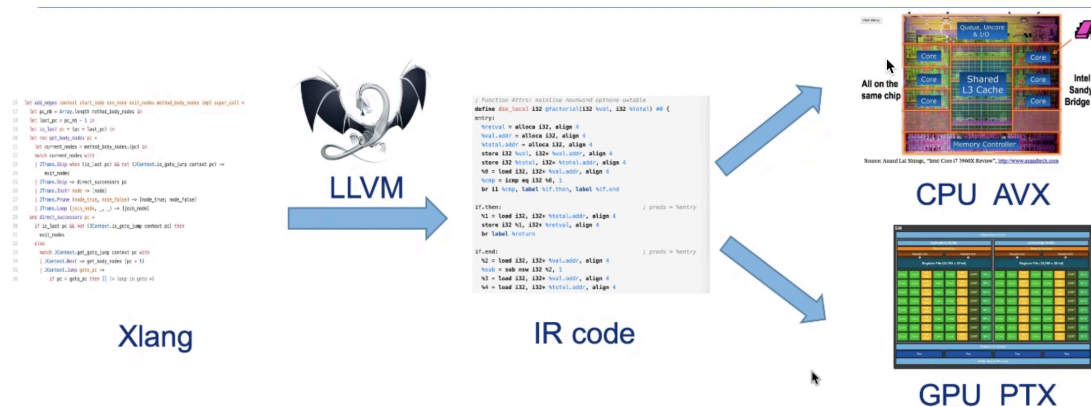


图 1: 三段式架构

理解了 LLVM 的基本架构我们就可以用它提供的 LLVM IR 来分析代码中是否有可重入函数。

### 3.2.2 用 IR 对函数进行分析

在这里我们需要分析 IR 中是否有不可重入函数。根据定义我们可以知道，不可重入函数有以下一条或多条特征：

- 使用了全局变量或静态变量
- 使用了 malloc 或者 new 开辟空间
- 调用了不可重入函数
- 返回值是全局或静态变量
- 使用了标准 I/O

因为标准 I/O 因为在实现的时候使用了全局数据结构，malloc 和 free 等函数使用了全局的内存分配表，所以视为不可重入函数。所以以上几点可以总结一下：不要使用全局或静态变量，以及不要在函数中调用其他的不可重入函数。

根据 LLVM 提供的接口，我们可以在编译的时候生成 bitcode，然后用 llvm-dis 命令将它变成 IR 代码。

而 LLVM 生成的中间代码中，全局变量或静态变量会带有 @ 标识符，而局部变量会带有 % 标识符。调用的外部函数也会有全局的 @ 标识符，所以我们评判的方法也是根据这些标识符来判断变量中是否有全局或静态变量，调用的函数是否为不可重入函数。

这就是检测不可重入函数的具体方法，后面我会先人工分析，再用编程实现自动分析。

## 4 实验结果

### 4.1 使用 ThreadSanitizer 工具检测多线程程序中潜在的 data races

#### 4.1.1 编写并行程序估算 $e$

在这里我使用 pthread，基本的思想是将 for 循环的所有循环体分配给我们创建的多个线程执行。

在这里我默认计算 1000 项级数（考虑到溢出的缘故，其实在后面的项对结果的影响已经接近于 0 了），并且设置线程数量为 8。估算  $e$  的代码 calculate\_e.c 如下：

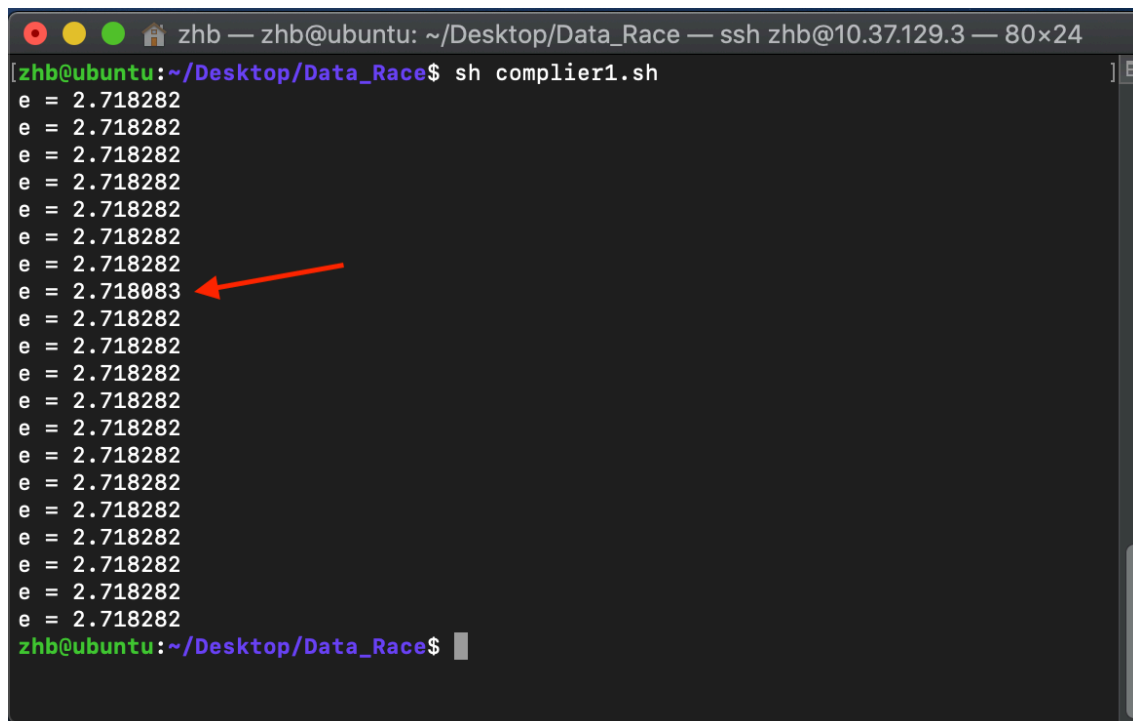
```

1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  double res = 1;
7  int count = 8; // 线程数量
8  int accuracy = 1000; // 计算精度，用多项式项数表示
9
10 void *Add(void *ptr) {
11     long thread_order = (int)ptr; // 第i个线程，用于表示线程序号
12     long n = thread_order; // 第n项
13
14     long loop_num = accuracy / count;
15     if(accuracy % count) loop_num++;
16     for(long j = 0; j < loop_num; j++){
17         n += j * count; // n % count = thread_order
18         long long denominator = 1;
19         for(int i = 2; i <= n; i++){
20             denominator *= i;
21         }
22         double term = 0;
23         if(denominator > 0) term = 1.0/denominator;
24         res += term;
25     }
26     return NULL;
27 }
28 int main(int argc, char **argv) {
29     if(argc == 3){
30         count = atoi(argv[1]);
31         accuracy = atoi(argv[2]);
32     }
33     pthread_t handle[count];
34     long i;
35     for(i = 1; i <= count; i++){
36
37         pthread_create(&handle[i - 1], NULL, Add, (void*)i);
38     }
39     for(int i = 1; i <= count; i++){

```

```
40     pthread_join(handle[i - 1], NULL);
41 }
42 printf("%f\n", res);
43 }
```

编写一个 sh 脚本，用命令 `clang calculate_e.c -o calculate_e -lpthread` 编译程序，并执行 20 次（脚本见 `complier1.sh` 文件），打开终端，用 `ssh` 连入虚拟机中的 Ubuntu 18.04（后面将解释为什么不使用 Mac 自带的 `clang`）用 `sh` 命令执行脚本，结果如下：



```
zhh — zhh@ubuntu: ~/Desktop/Data_Race — ssh zhh@10.37.129.3 — 80x24
[zhh@ubuntu:~/Desktop/Data_Race$ sh compier1.sh
e = 2.718282
e = 2.718282
e = 2.718282
e = 2.718282
e = 2.718282
e = 2.718282
e = 2.718282
e = 2.718083
e = 2.718282
e = 2.718282
e = 2.718282
e = 2.718282
e = 2.718282
e = 2.718282
e = 2.718282
e = 2.718282
e = 2.718282
e = 2.718282
e = 2.718282
zhh@ubuntu:~/Desktop/Data_Race$
```

图 2: 执行 20 次 `calculate_e.c` 的结果

上网查得， $e$  的近似值是 2.71828。

从结果我们可以看到，大部分时候，我们的程序可以进行正确的运算，但是在少数时候，如箭头标注处，计算结果会出错。

这种时候我们就需要使用 ThreadSanitizer 来对程序进行分析：

用命令 `clang -fsanitize=thread -g -O1 calculate_e.c -o tsan1`，再执行 `./tsan1`，得到结果如下：

```
zhib — zhib@ubuntu: ~/Desktop/Data_Race — ssh zhib@10.37.129.3 — 80x24
[zhib@ubuntu:~]$ cd Desktop/Data_Race/
[zhib@ubuntu:~/Desktop/Data_Race$ clang -fsanitize=thread -g -O1 calculate_e.c -o tsan1
[zhib@ubuntu:~/Desktop/Data_Race$ ./tsan1
=====
WARNING: ThreadSanitizer: data race (pid=10290)
  Write of size 8 at 0x0000006eeba0 by thread T1:
    #0 Add /home/zhib/Desktop/Data_Race/calculate_e.c:24 (tsan1+0x4b880d)

  Previous read of size 8 at 0x0000006eeba0 by thread T2:
    #0 Add /home/zhib/Desktop/Data_Race/calculate_e.c:24 (tsan1+0x4b8779)

  Location is global '<null>' at 0x000000000000 (tsan1+0x0000006eeba0)

  Thread T1 (tid=10292, running) created by main thread at:
    #0 pthread_create ??? (tsan1+0x427036)
    #1 main /home/zhib/Desktop/Data_Race/calculate_e.c:37 (tsan1+0x4b88f5)

  Thread T2 (tid=10293, running) created by main thread at:
    #0 pthread_create ??? (tsan1+0x427036)
    #1 main /home/zhib/Desktop/Data_Race/calculate_e.c:37 (tsan1+0x4b88f5)

SUMMARY: ThreadSanitizer: data race /home/zhib/Desktop/Data_Race/calculate_e.c:24
in Add
```

图 3: 用 ThreadSanitizer 分析 calculate\_e.c 的结果

不难看出, ThreadSanitizer 显示: 在源程序 37 行创建的线程在源 24 行的处出现了 data race。

于是回到源程序, 可以看到, 源程序 37 行是创建线程

```
pthread_create(&handle[i - 1], NULL, Add, (void*) i);
```

而源程序 24 行则是对全局变量 res 进行修改:

```
res += term;
```

很容易看出数据竞争的原因: 当两个线程同时访问全局变量 res 时, 第一个线程将 res 读到寄存器后, 在写回之前, res 又被另外一个线程读取到寄存器中, 这就造成了最后写回的结果少加了其中一项。

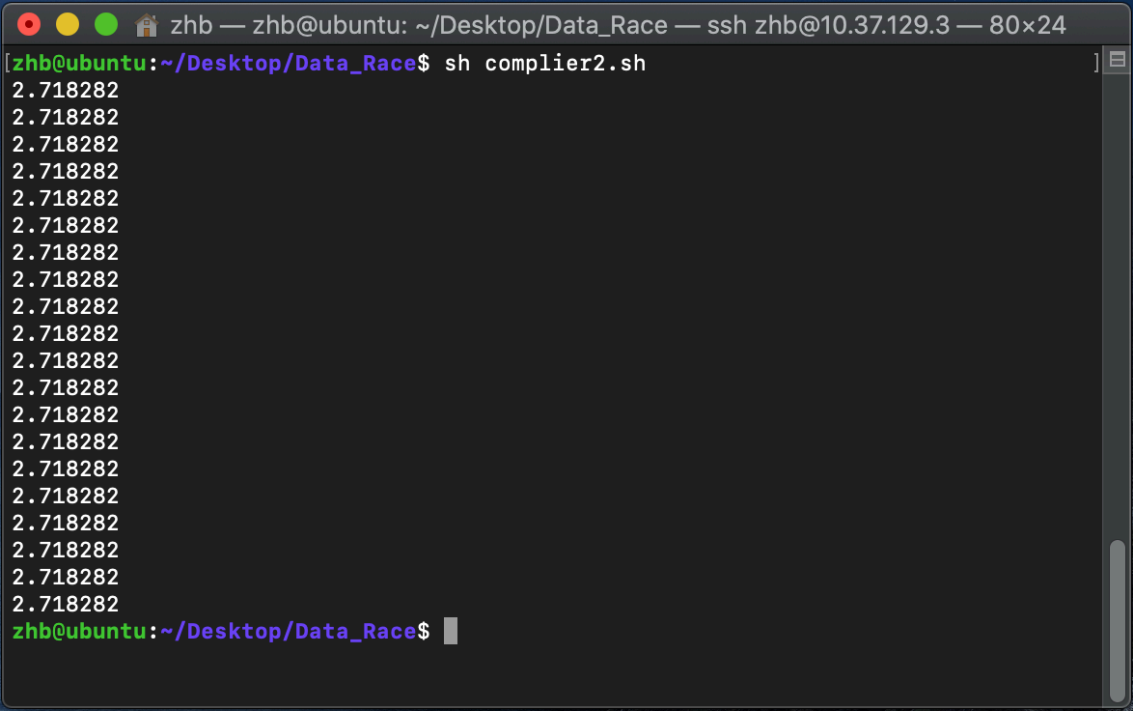
#### 4.1.2 用锁的方式来解决 data race

为了防止出现上面的情况, 我创建了 lock.c, 在 calculate\_e.c 的基础上, 在修改 res 变量的语句处使用了互斥锁, 它能保证能保证同一时间只有一个线程在执行临界区的代码, 此时其他线程只能等待。

代码如下：

```
pthread_mutex_lock(&lock);  
res += term;  
pthread_mutex_unlock(&lock);
```

然后再再写一个脚本，用 `clang lock.c -o lock -lpthread` 命令编译 `lock.c`，然后再执行 20 次（脚本见 `complier2.sh` 文件），结果如下：

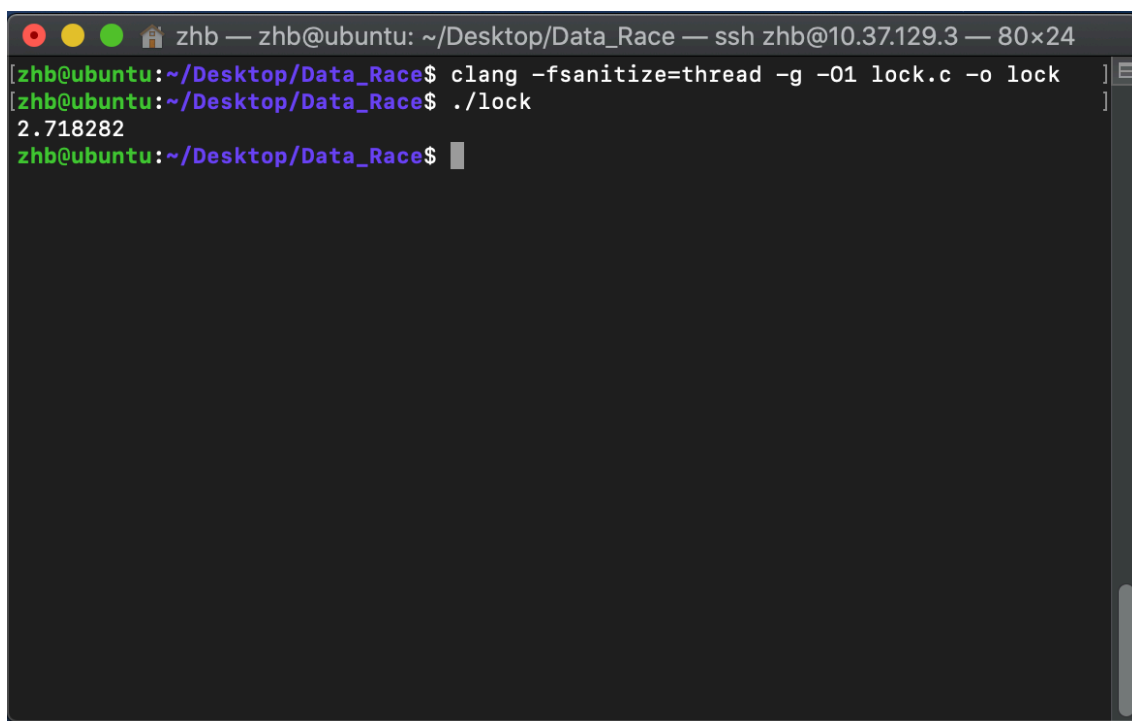
A terminal window titled 'zhh — zhh@ubuntu: ~/Desktop/Data\_Race — ssh zhh@10.37.129.3 — 80x24'. The prompt is 'zhh@ubuntu:~/Desktop/Data\_Race\$'. The user has entered 'sh compier2.sh'. The output consists of 20 lines, each displaying the number '2.718282'. The terminal has a dark background with light-colored text. The window title bar shows standard Ubuntu window controls (red, yellow, green buttons) and the title text. The prompt and command are in green, and the output is in white. The terminal is open in an SSH session from 10.37.129.3.

```
zhh@ubuntu:~/Desktop/Data_Race$ sh compier2.sh  
2.718282  
2.718282  
2.718282  
2.718282  
2.718282  
2.718282  
2.718282  
2.718282  
2.718282  
2.718282  
2.718282  
2.718282  
2.718282  
2.718282  
2.718282  
2.718282  
2.718282  
2.718282  
2.718282  
zhh@ubuntu:~/Desktop/Data_Race$
```

图 4: 执行 20 次 `lock.c` 的结果

可以看到这一次没有出现错误的结果。此时再用用 ThreadSanitizer 分析 `lock.c` 程序的执行情况：





```
zhh — zhh@ubuntu: ~/Desktop/Data_Race — ssh zhh@10.37.129.3 — 80x24
[zhh@ubuntu:~/Desktop/Data_Race$ clang -fsanitize=thread -g -O1 lock.c -o lock ]
[zhh@ubuntu:~/Desktop/Data_Race$ ./lock ]
2.718282
zhh@ubuntu:~/Desktop/Data_Race$
```

图 5: 用 ThreadSanitizer 分析 lock.c 的结果

可以看到并没有出现数据竞争的 warnings，说明此时不会出现 data races。  
至此，完成使用 ThreadSanitizer 检测多线程程序中潜在的数据竞争。

## 4.2 分析 LLVM 中间代码 IR 来判断是否有不可重入函数

### 4.2.1 人工分析

我编写了四个 pthread 程序，在线程函数中分别出现了全局变量、静态变量、malloc 和标准 I/O，用于代表主要的不可重入函数的情况，代码分别存在 Non-reentrant 目录下的 global、static、malloc 和 I/O 文件夹下。

首先打开 global 文件夹，用 clang -emit-llvm -c global.c -o global.bc 命令得到来 bitcode 文件，然后再用 llvm-dis global.bc 生成了 global.ll 文件。global.ll 就是 IR 代码。

global.c 的代码如下：

```
1 #include <pthread.h>
2
3 int global = 10;
```

```

4
5 void* fun(void* argv){
6     global = 20;
7     return NULL;
8 }
9 int main() {
10     pthread_t t;
11     pthread_create(&t, NULL, fun, NULL);
12     global = 43;
13     pthread_join(t, NULL);
14     return global;
15 }

```

而在 global.ll 文件中，对应函数 void\* fun(void\* argv) 的 IR 代码如下：

```

1 @global = global i32 10, align 4
2
3 ; Function Attrs: noinline nounwind optnone ssp uwtable
4 define i8* @fun(i8*) #0 {
5     %2 = alloca i8*, align 8
6     store i8* %0, i8** %2, align 8
7     store i32 20, i32* @global, align 4
8     ret i8* null
9 }

```

我们通过分析 fun 函数的 IR 代码很容易看到，在函数中的 global 带有全局标识符 @，所以很容易判断该函数是不可重入函数。

而对于 static.c，我用同样的方法进行编译，其中源程序中 void\* fun(void\* argv) 函数如下：

```

1 void* fun(void* argv){
2     int* res = (int*)argv;
3     static int x;
4     x = *res;
5     return NULL;
6 }

```

在 static.ll 中的 fun 函数的 IR 代码如下：

```

1 @fun.x = internal global i32 0, align 4
2

```

```

3 ; Function Attrs: noline nounwind optnone ssp uwtable
4 define i8* @fun(i8*) #0 {
5     %2 = alloca i8*, align 8
6     %3 = alloca i32*, align 8
7     store i8* %0, i8** %2, align 8
8     %4 = load i8*, i8** %2, align 8
9     %5 = bitcast i8* %4 to i32*
10    store i32* %5, i32** %3, align 8
11    %6 = load i32*, i32** %3, align 8
12    %7 = load i32, i32* %6, align 4
13    store i32 %7, i32* @fun.x, align 4
14    ret i8* null
15 }

```

可以看到，即使是在函数内部声明的静态变量，其 IR code 也会将其放在函数外部并声明该静态变量的归属，并且在函数中会给该变量加上全局标识符 @，如上文的 fun.x。

在 malloc.c 中，还是使用一样的方法编译，其中源代码中函数如下：

```

1 void* init(void*argv){
2     int* array = malloc(sizeof(int) * 10);
3     for(int i = 0; i < 10; i++)array[i] = 0;
4     free(array);
5     return argv;
6 }

```

生成的 malloc.ll 中对应函数如下：

```

1 define i8* @init(i8*) #0 {
2     %2 = alloca i8*, align 8
3     %3 = alloca i32*, align 8
4     %4 = alloca i32, align 4
5     store i8* %0, i8** %2, align 8
6     %5 = call i8* @malloc(i64 40) #3
7     %6 = bitcast i8* %5 to i32*
8     store i32* %6, i32** %3, align 8
9     store i32 0, i32* %4, align 4
10    br label %7
11
12    7:                                     ; preds = %15, %1
13    %8 = load i32, i32* %4, align 4
14    %9 = icmp slt i32 %8, 10

```

```

15     br i1 %9, label %10, label %18
16
17     10:                                     ; preds = %7
18     %11 = load i32*, i32** %3, align 8
19     %12 = load i32, i32* %4, align 4
20     %13 = sext i32 %12 to i64
21     %14 = getelementptr inbounds i32, i32* %11, i64 %13
22     store i32 0, i32* %14, align 4
23     br label %15
24
25     15:                                     ; preds = %10
26     %16 = load i32, i32* %4, align 4
27     %17 = add nsw i32 %16, 1
28     store i32 %17, i32* %4, align 4
29     br label %7
30
31     18:                                     ; preds = %7
32     %19 = load i32*, i32** %3, align 8
33     %20 = bitcast i32* %19 to i8*
34     call void @free(i8* %20)
35     %21 = load i8*, i8** %2, align 8
36     ret i8* %21
37 }

```

通过分析 IR 代码，我们可以发现其实在该函数中也有对 malloc 和 free 函数的调用，可以作为不可重入函数的标志。

最后是 IO.c: 函数的源代码如下：

```

1 void *fun(void *x) {
2     Global = 42;
3     printf("%d\n", Global);
4     return x;
5 }

```

生成的 IO.ll 中对应函数的部分如下：

```

1 ; Function Attrs: noinline nounwind optnone ssp uwtable
2 define i8* @fun(i8*) #0 {
3     %2 = alloca i8*, align 8
4     store i8* %0, i8** %2, align 8
5     store i32 42, i32* @Global, align 4

```

```

6   %3 = load i32, i32* @Global, align 4
7   %4 = call i32 @i8*, ... @printf(i8* getelementptr inbounds
8   ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 %3)
9   %5 = load i8*, i8** %2, align 8
10  ret i8* %5
11 }

```

不难发现，我们可以在 IR code 中找到对 printf 的调用，进而判断出它是不可重入函数。

#### 4.2.2 编程自动分析

根据我们人工分析的经验，对 IR code 进行分析的时候，我们可以按照一个个模块进行分析。而分析的角度有三个：全局变量、静态变量和函数中调用的不可重入函数（如库函数中的不可重入函数）。

IR code 中，使用了全局变量的特征是 @ 标志后面除了变量名外不会有其他内容，静态变量的特征是在 @ 后面的变量名会有一个 ' 来声明该静态变量属于那个函数，而在 IR code 中调用函数则会在 @ 和函数名的后面加上参数列表。

所以我的思路是这样子的：将一个函数作为一个模块分析。

- 对于全局变量：检测 @ 标识符后面是否有一个符合变量格式的变量名
- 对于静态变量：检测 @ 标识符后面是否有“函数名. 变量”的结构
- 对于调用的外部函数，首先提取出函数名，然后与不可重入函数向量中的函数名一一比较，如果在其中匹配则说明调用了不可重入函数。否则说明没有调用不可重入函数。

为了保证静态分析的可靠性，如果发现了一个不可重入函数则应将它放入不可重入函数向量中，以防其他函数调用它。

在这里我选择使用 C++ 来完成这个任务。我用类来抽象一个函数的 IR code：

```

1  class module{
2      string fun_name; // 函数名
3      string argument; // 参数列表
4      string ret_type; // 返回值类型
5      string content; // 函数体
6      vector<int> symbol; // 标志@在函数体的位置
7      vector<string> _global; // 函数中的全局变量
8      vector<string> _static; // 函数中的静态变量
9      vector<string> _non_reentrant_fun; // 函数中调用的不可重入函数
10 }

```

```

11 // 获取所有的@标志
12     void get_all_symbol();
13 // 检测symbol向量中是否有全局变量，有则放入__global向量
14     void detect_global();
15 // 检测symbol向量中是否有静态变量，有则放入__static向量
16     void detect_static();
17 // 检测symbol向量中是否有函数中调用的不可重入函数，有则放入__non_reentrant_fun向量
18     void detect_non_reentrant_fun();
19 // 对上述四种操作进行封装，若为不可重用函数则返回 true
20     bool detect();
21 public:
22 // 将IR code 中一个函数的字符串表示作为一个整体初始化模块类
23     module(string fun);
24 // 对detect() 进行封装并打印提示信息。
25     bool detect_and_print();
26 };

```

在实现具体操作的时候，首先将 IR code 以字符的形式读进 string。

```

1 string read_file(const char*file_name){
2     ifstream file(file_name);
3     stringstream ss;
4     ss << file.rdbuf();
5     string str = ss.str();
6     return str;
7 }

```

然后在全部代码中找到所有的 define，以此为依据来切分代码，将不同的函数存入全局变量向量 all\_function 中。

```

1 void divide(string s){
2     vector<int> pos;
3     int index = 0;
4     string sub = "define";
5     while ((index = s.find(sub, index)) < s.length())
6     {
7         pos.push_back(index);
8         string temp;
9         int flag = 0;
10        int target = 0;
11        for(int i = index; i < s.size(); i++){

```

```

12         if(s[i] == '}{'){
13             target = i;
14             flag = 1;
15             break;
16         }
17         if(flag) break;
18     }
19     temp = s.substr(index, target - index + 1);
20     all_function.push_back(temp);
21     index++;
22
23 }
24 }

```

然后按顺序，用各个函数模块的代码创建类，并用类中封装好的操作对代码进行分析，输出结果。由于库函数中不可重入函数过多，在这里我仅仅检测最常使用的 printf、scanf、malloc 和 free 函数。如果要检测其他函数只需要 push 进 Non\_reentrant\_fun 向量即可。

而传递文件名我使用了 main 的参数传递。

用 clang++ analyse.cpp -o analyse -std=c++11 对 analyse.cpp 进行编译，得到可执行文件。

于是我用之前生成的 IR code 进行测试：

首先是 global.ll，我在 fun 函数中使用了一个全局变量 global，运行程序，结果如下：

```

[zhb@zhanghb-MacBook-Pro analyse % ./analyse global.ll
Detect non reentrant function fun, details are as follows:
Use global valuable:global
zhb@zhanghb-MacBook-Pro analyse % █

```

图 6: 对 global.ll 进行分析

可以看到分析程序正确分析出了不可重入函数，并且正确提示了不可重入函数的原因。

然后是 static.ll，我在 fun 函数中使用了静态变量 x。运行程序，结果如下：

```
zhh@zhanghb-MacBook-Pro analyse % ./analyse static.ll
Detect non reentrant function fun, details are as follows:
Use static valuable:fun.x
zhh@zhanghb-MacBook-Pro analyse %
```

图 7: 对 static.ll 进行分析

可以看到分析函数正确分析出了不可重入函数，并提示了使用静态变量 fun.x。

接下来是 malloc.ll, 在该文件的 fun 函数中我使用了 malloc 和 free 两个不可重入函数。

```
zhh@zhanghb-MacBook-Pro analyse % ./analyse malloc.ll
Detect non reentrant function init, details are as follows:
Call non reentrant function:free malloc
zhh@zhanghb-MacBook-Pro analyse %
```

图 8: 对 malloc.ll 进行分析

可以看到成功分析出不可重入函数 malloc 和 free。

最后是 IO.ll, 在这里我在函数 init() 中既使用了全局变量 global, 又使用了标准输出 printf, 显然是不可重入函数。

分析的结果如下:

```
zhh@zhanghb-MacBook-Pro analyse % ./analyse IO.ll
Detect non reentrant function fun, details are as follows:
Call non reentrant function:printf
Use global valuable:Global
zhh@zhanghb-MacBook-Pro analyse %
```

图 9: IO.ll 进行分析

可以看到, 该程序可以正确检测到不可重入函数及其原因。

为了使说明更加严谨, 我写了一个不包含不可重入函数的程序 Control\_experiment.c, 并生成 IR code, 进行分析。

```
1 int add(int a, int b){
2     int c = a + b;
```



```
3     return c;  
4 }  
5 int main(){  
6     return 0;  
7 }
```

用 clang 生成 IR code 并进行分析:

```
[zhh@zhanghb-MacBook-Pro analyse % ./analyse Control_experiment.ll  
Non reentrant function not detected.  
zhh@zhanghb-MacBook-Pro analyse % █
```

图 10: 对 Control\_experiment.ll 进行分析

可以看到未检测到不可重入函数, 说明该程序正确运行。

#### 4.2.3 对这一部分实验的反思和总结

由于没有学过汇编原理, 所以我对 IR code 的分析显得很粗糙。事实上官方文档上似乎写着可以自己写 IR pass 来对程序进行分析, 但是这个超过了我的能力范围。

不过通过这部分实验, 我更加理解了 LLVM 架构的优势: 假设我们要写一个程序来完成一个任务, 而这个任务追求稳定性, 如信号处理, 我们就需要使用可重入函数。但是不同的人会使用不同的编程语言完成该任务, 如果我们对每种语言的源代码都开发语义分析程序来判断它是否为可重入函数, 那么这个工作量显然是巨大的。但是如果我们采用这种三段式的架构, 那么我们只需要对 IR code 进行分析就行了。而事实上如果要开发这样的程序似乎也并不困难, 我们只需要对是否采用全局变量、是否调用库函数进行分析就行了。这样的话也就可以用最少的工作量完成我们所需, 这也是 LLVM 的优势所在。

通过这一部分实验我也学到了很多, 虽然程序设计得并不是很好, 但是也收获颇丰。

## 5 遇到的问题及解决方法

这次实验主要遇到的问题是 Mac 的 clang 自带的 ThreadSanitizer 工具对互斥锁出现了 false positive, 就是在我加上了互斥锁之后, 明显就不会 data race 的情况下还是给我 data warning 的报错信息。

我在 stackoverflow 上面找了许久, 有人建议使用原子操作, 但是这超出了我的能力范围, 最终采用了使用 Ubuntu 下的 clang 来代替 Mac 的 clang, 最终解决了问题。

不过从这个问题也可以看出，ThreadSanitizer 工具对多线程程序的判断不一定是百分之百准确的，stackoverflow 上也有人吐槽这个工具并不是总是正确的，可能也它底层的实现方式有关。这也从侧面说明了多线程程序的复杂性。

还有一些奇奇怪怪的小问题，通过 Bing 和 Google 也顺利解决了。

## 参考文献

- [1] LLVM Language Reference Manual (<https://llvm.org/docs/LangRef.html>)
- [2] ThreadSanitizer (<http://clang.llvm.org/docs/ThreadSanitizer.html>)
- [3] Why does ThreadSanitizer report a race with this lock-free example?  
(<https://stackoverflow.com/questions/37552866/why-does-threadsanitizer-report-a-race-with-this-lock-free-example>)