并行与分布式计算作业

第三次作业

姓名：施天予

班级：计科2班

学号：19335174

一、问题描述

简答题

1. 并行编程主要有哪些模型，各自的特点是什么?
2. 多线程编程中，可重入和线程安全的定义以及关系是什么?
3. 将以下函数改成可重复函数;

```
int i;
int fun1()
{
    return i * 5;
}
int fun2()
{
    return fun1() * 5;
}
```

编程题

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，并对比加上锁以后检测结果的变化，分析及给出案例程序并提交分析报告。

基本思路

1. 编写具有数据竞争的多线程程序（ C或者Java）；
2. 利用LLVM或者Soot将C或者Java程序编译成字节码;
3. 利用LLVM或者soot提供的数据竞争检测工具检测;
4. 对有数据竞争的地方加锁，观察检测结果;

参考： http://clang.llvm.org/docs/ThreadSanitizer.html。

二、解决方案

简答题

问题1

并行编程主要有哪些模型，各自的特点是什么？

- **消息传递**
  - 显式点对点
  - 封装本地数据的独立任务
  - 任务通过交换消息进行交互
- **共享内存**
  - 任务共享一个公共地址空间
  - 任务通过异步读写该空间进行交互（读写共享变量）
- **数据并行**
  - 没有严格的通信和同步
  - 任务执行一系列独立的操作
  - 数据通常跨任务分配
  - 也称为“尴尬并行”

问题2

多线程编程中，可重入和线程安全的定义以及关系是什么？

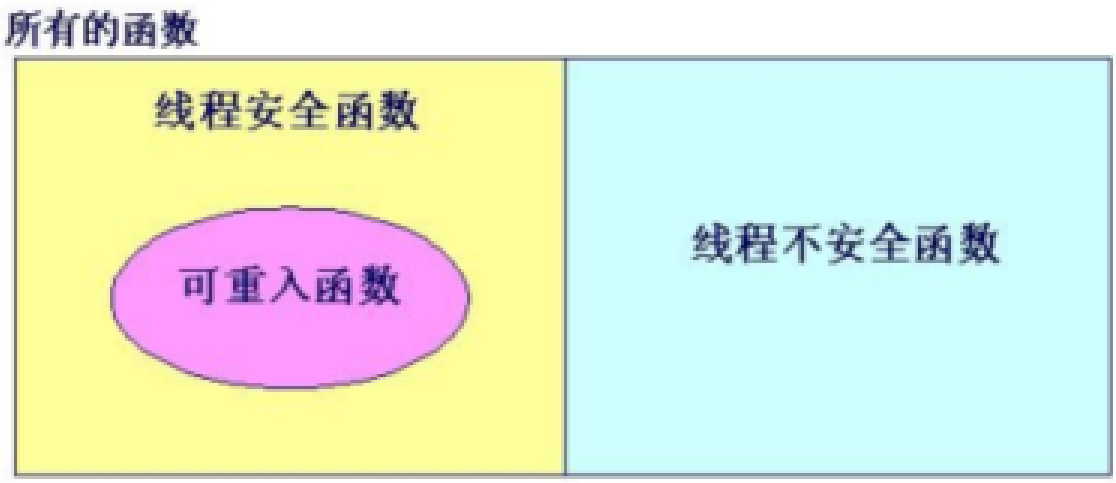
定义

- **可重入**
  - 可重入：多个执行流反复执行一个代码，其结果不会发生改变，通常访问的都是各自的私有栈资源。
  - 可重入函数：当一个执行流因为异常或者被内核切换而中断正在执行的函数而转为另外一个执行流时，当后者的执行流对同一个函数的操作并不影响前一个执行流恢复后执行函数产生的结果。
  - 可重入函数满足的条件
    - 不使用全局变量或静态变量
    - 不使用malloc或者new开辟出的空间
    - 不调用不可重入函数
    - 不返回静态或全局数据，所有数据都由函数的调用者提供
    - 使用本地数据，或者通过制作全局数据的本地拷贝来保护全局数据
    - 不调用标准I/O
- **线程安全**

  多线程访问同一代码，能够正确执行，不会产生不确定的结果。采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取结束并且释放了锁，其他线程才可使用，保证了数据的一致性。

关系

函数可以是可重入的，也可以是线程安全的，或者两者皆是，或者两者皆非。不可重入函数不能由多个线程使用。



1. 线程安全是在多线程情况下引发的，而可重入函数可以在只有一个线程的情况下发生。
2. 如果一个函数有全局变量，则这个函数既不是线程安全也不是可重入的。
3. 如果一个函数当中的数据全是自己私有栈空间的，则这个函数既是线程安全也是可重入的。
4. 如果将对临界资源的访问加锁，则这个函数是线程安全的；但如果重入函数的话加锁还未释放，则会产生死锁，因此不能重入。
5. 线程安全函数能够使不同的线程访问同一块地址空间，而可重入函数要求不同的执行流对数据的操作不影响结果，使结果是相同的。

问题3

将以下函数改成可重复函数；

```
int i;
int fun1()
{
    return i * 5;
}
int fun2()
{
    return fun1() * 5;
}
```

修改后的代码如下

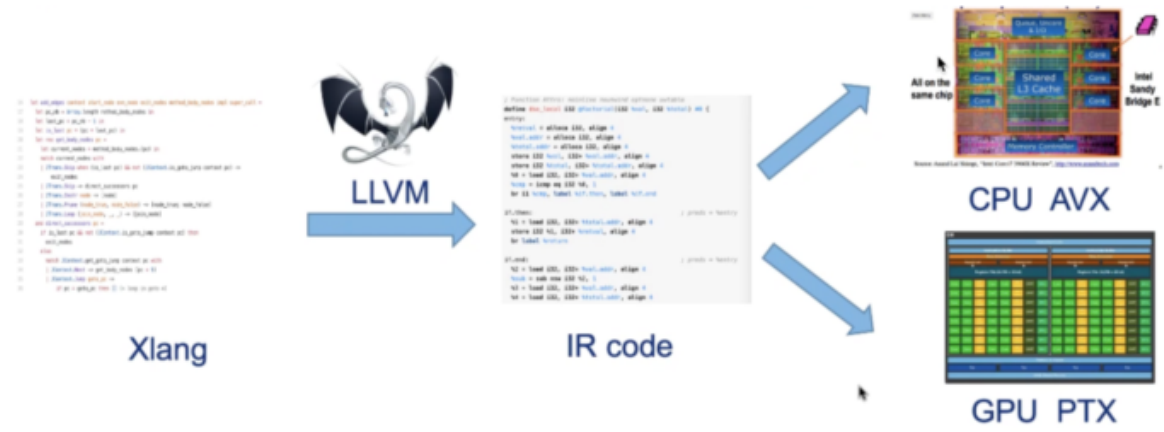
```c
int fun1(int i)
{
    return i * 5;
}
int fun2(int i, int (*fun1)(int))
{
    return fun1(i) * 5;
}
```

编程题

LLVM与clang

LLVM 是以 BSD 许可来开发的开源的编译器框架系统，基于 C++ 编写而成，利用虚拟技术来优化以任意程序语言编写的程序的编译时间、链接时间、运行时间以及空闲时间，最早以 C/C++ 为实现对象，对开发者保持开放，并兼容已有脚本。

Clang 是一个 C、C++、Objective-C 和 Objective-C++ 编程语言的编译器前端，采用底层虚拟机（LLVM）作为后端。Clang 是一个高度模块化开发的轻量级编译器，编译速度快、占用内存小、有着友好的出错提示。



clang编译的过程

clang是一个 C、C++和 Objective-C的编译器，编译过程如下：

处理 -> 语法解析 -> 代码生成&优化 -> 汇编 -> 链接
.c -> AST -> .s -> .o -> .out

```
sty@ubuntu:~$ clang -ccc-print-phases test.c -o test
            +- 0: input, "test.c", c
         +- 1: preprocessor, {0}, cpp-output
      +- 2: compiler, {1}, ir
   +- 3: backend, {2}, assembler
+- 4: assembler, {3}, object
5: linker, {4}, image
```

- **预处理**：这个阶段会对输入的源文件进行标记化处理、宏扩展、#include扩展和其他预处理器指令的处理。输入格式：.c、.cpp、.m、.mm；输出格式：.i、.ii、.mi、.mii。
- **语法解析**：这个阶段会解析输入文件，将预处理标记转换为解析树。一旦以解析树的形式出现，它也会用语义分析来计算表达式的类型，确认代码格式是否正确，该阶段负责生成大多数的编译警告和错误。输出格式：AST(抽象语法树，Abstract Syntax Tree)。
- **代码生成和优化**：这个阶段会将AST转换为底层的中间代码(称为 LLVM IR)，再最终转换为机器码.该阶段负责优化生成的代码, 并处理特定目标的代码生成。输出格式：.s、汇编文件。
- **汇编**：这个阶段将编译器的输出转换为目标文件。输出格式：.o、object文件。
- **链接**：这个阶段会将多个目标文件合并为一个可执行文件或动态库。输出格式：.out、.so。

实验步骤

1. 因为 ThreadSanitizer 工具不支持在 Windows 下运行，所以我在 Ubuntu 系统中安装Clang 编译器对案例程序进行编译和运行，利用 Ubuntu 中 apt 软件包管理工具可完成安装。
2. 编写程序使得两个线程运行两个不同的函数在同一时间对同一全局变量进行修改。
3. 利用 LLVM 将程序编译成 IR code。
4. 利用 LLVM 提供的数据竞争检测工具 ThreadSanitizer 自动生成分析，观察数据竞争的信息。
5. 对有数据竞争的地方加锁，再次利用 LLVM 观察检测结果。

三、实验结果

环境配置与安装

利用Ubuntu的apt软件包管理工具，在终端输入 `sudo apt install clang` 安装Clang编译器，完成后输入 `clang -v` 检查安装是否成功。

```
sty@ubuntu:~$ clang -v
clang version 10.0.0-4ubuntu1
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
Found candidate GCC installation: /usr/bin/../lib/gcc/x86_64-linux-gnu/9
Found candidate GCC installation: /usr/lib/gcc/x86_64-linux-gnu/9
Selected GCC installation: /usr/bin/../lib/gcc/x86_64-linux-gnu/9
Candidate multilib: .;@m64
Selected multilib: .;@m64
```

说明clang安装成功。

数据竞争的多线程程序

先编写一个数据竞争的多线程程序，程序代码如下：

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int Global;
void *Thread1(void *x) {
    Global++;
    return NULL;
}
void *Thread2(void *x) {
    Global--;
    return NULL;
}

int main() {
    pthread_t *thread_handles =
(pthread_t*)malloc(2*sizeof(pthread_t));
    pthread_create(&thread_handles[0], NULL, Thread1, NULL);
    pthread_create(&thread_handles[1], NULL, Thread2, NULL);
    pthread_join(thread_handles[0], NULL);
    pthread_join(thread_handles[1], NULL);
    free(thread_handles);
    return 0;
}
```

在终端输入 `clang -S -emit-llvm test.c -o test.ll` 可得到test.c的字节码。全局变量Global和两个函数的字节码如下：

```
@Global = common dso_local global i32 0, align 4

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i8* @Thread1(i8* %0) #0 {
  %2 = alloca i8*, align 8
```

```
  store i8* %0, i8** %2, align 8
  %3 = load i32, i32* @Global, align 4
  %4 = add nsw i32 %3, 1
  store i32 %4, i32* @Global, align 4
  ret i8* null
}

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i8* @Thread2(i8* %0) #0 {
  %2 = alloca i8*, align 8
  store i8* %0, i8** %2, align 8
  %3 = load i32, i32* @Global, align 4
  %4 = add nsw i32 %3, -1
  store i32 %4, i32* @Global, align 4
  ret i8* null
}
```

可发现一些规律：

- 全局变量以 `@` 开头
- 外部函数以 `define` 开头

显然Thread1和Thread2是会对全局变量Global产生数据竞争的。利用ThreadSanitizer，在终端输入

```
clang -fsanitize=thread -g -o1 test.c -o test
./test
```

将程序编译和运行，利用数据竞争检测工具观察结果

```
sty@ubuntu:~$ clang -fsanitize=thread -g -o1 test.c -o test
sty@ubuntu:~$ ./test
==================
WARNING: ThreadSanitizer: data race (pid=37533)
  Write of size 4 at 0x000000f12358 by thread T2:
    #0 Thread2 /home/sty/test.c:11:11 (test+0x4b194e)

  Previous write of size 4 at 0x000000f12358 by thread T1:
    #0 Thread1 /home/sty/test.c:7:11 (test+0x4b18ee)

  Location is global 'Global' of size 4 at 0x000000f12358 (test+0x000000f12358)

  Thread T2 (tid=37536, running) created by main thread at:
    #0 pthread_create <null> (test+0x424a2b)
    #1 main /home/sty/test.c:18:5 (test+0x4b19e3)

  Thread T1 (tid=37535, finished) created by main thread at:
    #0 pthread_create <null> (test+0x424a2b)
    #1 main /home/sty/test.c:17:5 (test+0x4b19ba)

SUMMARY: ThreadSanitizer: data race /home/sty/test.c:11:11 in Thread2
==================
ThreadSanitizer: reported 1 warnings
```

发现出现了 `data race` 的报错，发生了数据竞争，并指出竞争发生在第7行（Thread1）和第11行（Thread2）

```
int Global;
void *Thread1(void *x) {
    Global++;
    return NULL;
}
void *Thread2(void *x) {
    Global--;
    return NULL;
}
```

确实对全局变量Global的加和减会产生数据竞争，与预期相符。

加锁解决数据竞争

使用pthread的互斥锁，在修改Global变量的同一时间只有一个线程能执行临界区代码，修改后的程序代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t mutex;
int Global;

void *Thread1(void *x) {
    pthread_mutex_lock(&mutex);
    Global++;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
void *Thread2(void *x) {
    pthread_mutex_lock(&mutex);
    Global--;
    pthread_mutex_unlock(&mutex);
    return NULL;
}

int main() {
    pthread_t *thread_handles =
(pthread_t*)malloc(2*sizeof(pthread_t));
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&thread_handles[0], NULL, Thread1, NULL);
    pthread_create(&thread_handles[1], NULL, Thread2, NULL);
    pthread_join(thread_handles[0], NULL);
    pthread_join(thread_handles[1], NULL);
    pthread_mutex_destroy(&mutex);
```

```
    free(thread_handles);
    return 0;
}
```

在终端输入 `clang -S -emit-llvm test.c -o test.ll` 可得到test.c的字节码。全局变量Global和两个函数，以及相关pthread互斥锁的字节码如下：

```
@mutex = common dso_local global %union.pthread_mutex_t
zeroinitializer, align 8
@Global = common dso_local global i32 0, align 4

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i8* @Thread1(i8* %0) #0 {
  %2 = alloca i8*, align 8
  store i8* %0, i8** %2, align 8
  %3 = call i32 @pthread_mutex_lock(%union.pthread_mutex_t* @mutex)
#3
  %4 = load i32, i32* @Global, align 4
  %5 = add nsw i32 %4, 1
  store i32 %5, i32* @Global, align 4
  %6 = call i32 @pthread_mutex_unlock(%union.pthread_mutex_t*
@mutex) #3
  ret i8* null
}

; Function Attrs: nounwind
declare dso_local i32 @pthread_mutex_lock(%union.pthread_mutex_t*)
#1

; Function Attrs: nounwind
declare dso_local i32
@pthread_mutex_unlock(%union.pthread_mutex_t*) #1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i8* @Thread2(i8* %0) #0 {
  %2 = alloca i8*, align 8
  store i8* %0, i8** %2, align 8
  %3 = call i32 @pthread_mutex_lock(%union.pthread_mutex_t* @mutex)
#3
  %4 = load i32, i32* @Global, align 4
  %5 = add nsw i32 %4, -1
  store i32 %5, i32* @Global, align 4
  %6 = call i32 @pthread_mutex_unlock(%union.pthread_mutex_t*
@mutex) #3
  ret i8* null
}
```

再次利用ThreadSanitizer，在终端输入

```
clang -fsanitize=thread -g -o1 test2.c -o test2
./test2
```

观察结果发现没有WARNING，说明此时没有`data race`，数据竞争问题解决。

```
sty@ubuntu:~$ clang -fsanitize=thread -g -o1 test2.c -o test2
sty@ubuntu:~$ ./test2
sty@ubuntu:~$
```

四、遇到的问题及解决方法

这次实验遇到的问题不多，主要也是对clang的安装和编译的的一些命令不太了解，好在查阅参考文档并上网搜索都能顺利解决。

这次实验也让我收获了很多。ThreadSanitizer 这个线程检测工具，能够检测数据竞争、 死锁等问题。它在程序执行期间，会一直检测，并维护一个全局的事件队列。每当一个新的事件发生时， ThreadSanitizer 会将其加入到事件队列中，并更新自己的状态，然后通过状态判断这个事件是否导致了各类数据竞争.总而言之，数据竞争会导致数据的不确定性，多线程编程过程中需要注意对数据的保护，如加锁等使线程同步的方法。