实验4 程序构建和软件测试

4.0 Overview 项目任务

- 使用Make构建动物园程序
- 使用gtest进行程序测试
- 使用valgrind进行内存泄漏检测
- 使用gprof可视化调用链并分析恶意占用CPU的代码
- 使用perf进行CPU占用检测并绘制火焰图

4.0.1 项目代码

由老师在课上分发。

4.1 Make使用

4.1.1 实验目的

了解makefile的基础规则，使用make构建动物园程序

4.1.2 实验内容与要求

独立完成实验内容，对实验步骤进行截图，并认真撰写实验报告。

要求

- 掌握makefile变量的使用
- 掌握makefile的函数
- 掌握makefile的静态规则
- 掌握makefile的伪目标
- 掌握makefile的撰写

内容

一、makefile介绍

makefile规定了整个工程的编译规则。一个工程中的源文件不计其数，并且按类型、功能、模块分别放在若干个目录中，makefile定义了一系列的规则来指定，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作。makefile中可以指定执行各种shell命令。

makefile带来的好处就是——“自动化编译”，一旦写好，只需要一个make命令，整个工程完全自动编译，极大地提高了软件开发效率。 make是一个命令工具，是一个解释makefile中指令的命令工具，一般来说，大多数的IDE都有这个命令，比如：Delphi的make，Visual C++的nmake，Linux下GNU的make。可见，makefile都成为了一种在工程方面的编译方法。

二、makefile规则

以下是make的规则的格式：

```
<target> ... : <prerequisites> ...
    <command>
    ...
```

- target: 可以是一个目标文件(*.o)，也可以是一个执行文件，还可以是一个标签（label）。对于标签这种特性，在后续的“伪目标”章节中会有叙述。
- prerequisites: 生成该target所依赖的文件或其它target的集合。
- command: 该target要执行的命令。

这是一个文件的依赖关系，也就是说，target这一个或多个的目标文件依赖于prerequisites中的文件，其生成规则定义在commands中。说白一点就是:

> prerequisites中如果有一个以上的文件比target文件要新的话，command所定义的命令就会被执行。

这就是makefile的规则，也就是makefile中最核心的内容。

make并不管命令是怎么工作的，他只管执行所定义的命令。make会比较targets文件和prerequisites文件的**修改日期**，如果prerequisites文件的日期要比targets文件的日期要新，或者target不存在的话，那么，make就会执行后续定义的命令。

当我们在shell中输入`make`后，make找到当前目录下的Makefile（默认为Makefile或makefile），找到其中的第一个target，并执行对应的commands。

在本次实验中，以下代码即makefile中的第一个target，其中的符号和自动变量在之后的章节中会讲解。

```
${BIN_TARGET}: ${OBJECT}
   ${CXX} -o $@ $^ ${CXXFLAGS}
```

三、makefile的基础语法

1. 变量：make中使用`变量名 = 变量值`来定义变量，使用`$(变量名)`或`${变量名}`来获得变量名中保存的变量值。我们代码中开头的一堆`XDIR`即使用了变量定义。`CXXFLAGS`中的`-I${IDIR}`使用了变量展开，即展开为`-I./include`。

   ```
   IDIR = ./include
   ODIR = ./obj
   SDIR = ./src
   BDIR = ./bin
   TDIR = ./test
   SCRIPT_DIR = ./script

   CXX = g++
   CXXFLAGS = -I${IDIR} -lgtest -lpthread -std=c++14 -g -pg
   ```

2. 函数：makefile内置了很多功能函数，主要用来简化makefile中的字符串操作。函数的调用，很像变量的使用，也是以 `$` 来标识的，其语法如下：

   ```
   $(<function> <arguments>)
   ```

   这里， `<function>` 是函数名。`<arguments>` 为函数参数列表。参数间以逗号 `,` 分隔，而函数名和参数之间以“空格”分隔。函数调用以 `$` 开头，以圆括号或花括号把函数名和参数括起。函数中的参数可以使用变量。

   以下是几个常用的makefile函数：

   - wildcard：通配符**扩展**函数

     ```
     $(wildcard PATTERN...)
     ```

     该函数用法如上，函数自动匹配通配符并展开为已经存在的、使用空格分开的、匹配此模式的所有文件列表。

     本实验代码中的用法如下：

     ```
     # SOURCE使用wildcard函数匹配SRC目录下的.cpp文件
     SOURCE = $(wildcard ${SDIR}/*.cpp)
     ```

   - notdir：去文件名函数（去掉目录部分，只保留文件名）

     ```
     $(notdir <names...>)
     ```

     从文件名序列 `<names>` 中取出非目录部分。非目录部分是指最后一个反斜杠（ `/` ）之后的部分。返回文件名序列 `<names>` 的非目录部分。

   - patsubst：通配符**替换**函数

     ```
     $(patsubst pattern, replacement, text)
     ```

     该函数查找text中的单词（单词以“空格”、“Tab”或“回车”“换行”分隔）是否符合模式pattern，如果匹配的话，则以replacement替换。

     本实验代码中的用法如下：

     ```
     # OBJECT使用patsubst函数将SOURCE对应的.cpp文件的后缀替换成.o
     OBJECT = $(patsubst %.cpp, ${ODIR}/%.o, $(notdir ${SOURCE}))
     ```

3. 自动变量：可以理解为由 Makefile 自动产生的变量。

   在模式规则中，规则的目标和依赖的文件名代表了一类的文件。在 Makefile 中描述规则时，依赖文件和目标文件往往是变动的，在命令中不应该出现具体的文件名称，否则规则将失去意义。那么如何表示这些文件呢？使用“自动化变量”！自动化变量的取值，取决于执行规则的目标文件和依赖文件。

   下面对部分自动化变量进行说明。

   1. `$@` ：规则目标的**文件名**。

2. `$^`：代表的是所有依赖文件列表。
3. `$<`：依赖对象集合中的第一个文件。

本实验代码中的用法如下：

```
# 第二行可扩展为
# GCC -o BIN_TARGET的值 OBJECT中的所有文件 CXXFLAG中的所有选项
${BIN_TARGET}: ${OBJECT}
    ${CXX} -o $@ $^ ${CXXFLAGS}
```

四、makefile中的隐式规则

make包含有一些内置的或者隐含的规则，这些规则定义了如何从不同的依赖文件建立特定类型的目标。隐式规则能够告诉make怎么样使用传统的技术完成任务，这样用户在使用它们的话就不必详细指定编译器的具体细节，而只需要把目标文件列出即可。make会自动搜索隐式规则来确定如何生成目标文件。

常见的隐式规则如下：

隐含规则	等效命令
“.c”编译为“.o”	`$(CC) -c $(CPPFLAGS) $(CFLAGS)`
“.cc”或“.C”编译为“.o”	`$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)`

例如以下makefile使用显示规则撰写：

```
objects=main.o add.o dec.o mul.o div.o
pragram:${objects}
    gcc $^ -o $@
main.o:main.c
    gcc $^ -o $@
add.o:add.c
    gcc $^ -o $@
dec.o:dec.c
    gcc $^ -o $@
mul.o:mul.c
    gcc $^ -o $@
div.o:div.c
    gcc $^ -o $@
clean:
    rm *.o pragram
```

现将上面定义的显式规则改为隐式规则：

```
objects=main.o add.o dec.o mul.o div.o
CC=gcc
CFLAGS=-Wall -O -g
pragram:$(objects)
   ${CC} $^ -o $@
clean:
   rm *.o pragram
```

相比于显式规则，隐式规则在书写上简洁很多，在大型项目中更具优势。

五、makefile中的静态模式

静态模式可以更加容易地定义多个目标的规则，可以让我们的规则变得更加有弹性和灵活。我们还是先来看一下语法：

```
<targets ...> : <target-pattern> : <prereq-patterns ...>
   <commands>
    ...
```

- targets：定义了一系列的目标文件，可以有通配符。是目标的一个集合。
- target-pattern：是指明了targets的模式，也就是的目标集模式。
- prereq-patterns：是目标的依赖模式，它对target-pattern形成的模式再进行一次依赖目标的定义。

这样描述这三个东西，可能还是没有说清楚，还是举个例子来说明一下吧。如果我们的`<target-pattern>`定义成`%.o`，则代表`<target>`集合中所有以`.o`结尾的`<target>`。而如果我们的`<prereq-patterns>`定义成`%.c`，则是对`<target-pattern>`所形成的目标集进行二次定义，其计算方法是，取`<target-pattern>`模式中的`%`（也就是`%.o`中去掉了`.o`这个结尾）代表的部分，并为其加上`.c`这个结尾，形成的新集合。在多数时候，`<target>`是可以省略的，以下是一个例子：

```
%.o : %.c
   gcc -c $< -o $@
```

本实验代码中的用法如下：

```
# "静态模式+自动化变量" 来生成 ".o" 目标文件
# $@代表target的名字，$<代表右侧输入的第一个名字
# 该规则展开如下：
# ./obj/foo.o : ./src/foo.c
#    $(CXX) -c $(CXXFLAGS) foo.c -o foo.o
$(ODIR)/%.o: $(SDIR)/%.cpp
   $(CXX) -c $(CXXFLAGS) $< -o $@
```

六、makefile中的伪目标

我们的程序代码中，我们提到过一个“clean”的目标，这是一个“伪目标”，

```
clean:
  #find $(ODIR) -name *.o -exec rm -rf {} \; #这个是find命令，不懂的可以查下资料
  rm -f $(ODIR)/*.o
  rm -rf $(BDIR)/*.out
  rm -f $(TDIR)/*.out $(TDIR)/*.txt $(TDIR)/*.png $(TDIR)/*.out
  rm -rf gmon.out
```

正像我们前面例子中的“clean”一样，既然我们生成了许多文件编译文件，我们也应该提供一个清除它们的“目标”以备完整地重编译而用。（以“make clean”来使用该目标）

因为，我们并不生成“clean”这个文件。“伪目标”并不是一个文件，只是一个标签，由于“伪目标”不是文件，所以make无法生成它的依赖关系和决定它是否要执行。我们只有通过显式地指明这个“目标”才能让其生效。当然，“伪目标”的取名不能和文件名重名，不然其就失去了“伪目标”的意义了。

当然，为了避免和文件重名的这种情况，我们可以使用一个特殊的标记“.PHONY”来显式地指明一个目标是“伪目标”，向make说明，不管是否有这个文件，这个目标就是“伪目标”。

```
.PHONY : clean
```

伪目标一般没有依赖的文件。但是，我们也可以为伪目标指定所依赖的文件。伪目标同样可以作为“默认目标”，只要将其放在第一个。一个示例就是，如果你的Makefile需要一口气生成若干个可执行文件，但你只想简单地敲一个make完事，并且，所有的目标文件都写在一个Makefile中，那么你可以使用“伪目标”这个特性。

七、完整的代码

可参见该项目的Makefile，在项目目录下使用`make`来构建该项目。

4.2 Googletest使用

4.2.1 实验目的

了解Googletest，编写简单的gtest测试程序

4.2.2 实验内容与要求

独立完成实验内容，对实验步骤进行截图，并认真撰写实验报告。

要求

- 掌握Gtest的常用断言
- 掌握Gtest常用的测试手段
- 编写Gtest函数

内容

一、Googletest基础

Googletest（通常简称为gtest）是由Google开发的一个C++单元测试框架，用于编写和运行自动化测试。它支持各种测试方法，包括单元测试、集成测试和功能测试等，并提供了丰富的测试工具和测试报告，以帮助开发人员快速、准确地测试和验证代码的正确性和健壮性。

使用GoogleTest时，首先要编写断言，这些断言是检查条件是否为真的语句。断言的结果可以是成功、非致命的失败或致命的失败。如果发生致命故障，则中止当前功能；否则程序正常继续。

常见定义：

- Test Case：Test Case是指一个独立的测试用例，通常测试一个特定的函数或方法。每个Test Case都是使用`TEST`或`TEST_F`宏定义的，可以独立运行和测试。
- Test Suite：Test Suite是指一组相关的测试用例，通常用于打包测试一个共同的功能或对象。一个Test Suite可以包含多个Test Case。
- Test Fixture：当一个Test Suite中的多个Test Case需要共享公共的对象和子例程时，可以使用Test Fixture来封装共享部分。在使用Test Fixture时，必须使用`TEST_F`宏，其它情况则使用`TEST`宏。

二、assertion、test suite和test fixture

1. assertion：“断言”是类似于函数调用的宏，用于判定类或函数的行为是否达到预期。当断言失败时，GoogleTest会打印一条失败消息。以下两类常见的断言：
    1. ASSERT_*：当断言失败时，产生致命错误、并终止当前函数，包括：ASSERT_TRUE、ASSERT_FALSE、ASSERT_EQ、ASSERT_NE、ASSERT_STREQ、ASSERT_STRNE等。
    2. EXPECT_*：当断言失败时，产生非致命错误，并不会终止当前函数。
2. Test Suite：Test Suite是测试的基础，其包含一个或多个Test Case。应将逻辑上具有相关性的测试用例放到一个Test Suite中。以下是一个Test Case的格式

    ```
    TEST(TestSuiteName, TestName) {
      ... test body ...
    }
    ```

    - 使用TEST()宏来定义和命名Test Case。每个Test Case可视为一个不返回值的普通C++函数。
    - 在这个函数中，连同您想要包含的任何有效的C++语句一起，使用各种GoogleTest断言来检查值。
    - 测试的结果由断言决定；如果测试中的任何断言失败（无论是致命的还是非致命的），亦或测试崩溃，则整个测试失败。否则成功。

```
// 本代码中，MouseTest是一个Test Suite，包含两个Test Case（TrueName和FalseName），用于测试mouse.Name()是否返回Mouse。
TEST(MouseTest, TrueName)
{
   Mouse mouse;
   EXPECT_EQ("Mouse", mouse.Name());
}

TEST(MouseTest, FalseName)
{
   Mouse mouse;
   EXPECT_EQ("Cat", mouse.Name());
}
```

3. Test Fixture：当一个Test Suite中的多个测试需要共享公共对象和子例程时，可以使用Test Fixture对Test Suite进行包装，以使用相同的数据配置进行不同的Test Case测试。其使用方法如下：
   1. 从`::testing::Test`派生一个类。它的主体以`protected:`开头，因为我们希望从子类访问fixture的成员。
   2. 在类内部，声明您计划使用的任何对象。
   3. 可实现虚函数`virtual void SetUp()`，用于Test Fixture中每个Test Case的测试环境的初始化。（在每个Test Case中最先执行。）
   4. 可实现虚函数`virtual void TearDown()`，用于Test Fixture中每个Test Case的测试环境的资源清理。（在每个Test Case中最后执行。）
   5. 如果需要，可定义一些测试中需要用到的公共的子例程。

   使用Test Fixture时，请使用`TEST_F()`而不是`TEST()`，因为它允许您访问Test Fixture中的对象和子例程：

```
TEST_F(TestFixtureClassName, TestName) {
   ... test body ...
}
```

   与`TEST()`不同，在`TEST_F()`中，第一个参数必须是Test Fixture的名称（_F代表“Fixture”）。可认为Test Suite和Test Fixture同名。

   不幸的是，C++宏系统不允许我们创建一个可以同时处理这两种类型测试的宏。使用错误的宏会导致编译器错误。
   此外，在`test_F()`中使用Test Fixture之前，您必须**首先定义它**，否则您将得到编译器错误“虚拟外部类声明”。

   对于用`test_F()`定义的每个Test Case，GoogleTest将在运行时创建一个新的Test Fixture，并立即通过`SetUp()`对其进行初始化，然后运行测试，再调用`TearDown()`进行清理，最后删除该Test Fixture。**请注意**：同一Test Suite中的不同Test Case具有不同的Test Fixture对象。GoogleTest不会对多个Test Case重复使用同一个Test Fixture，即一个Test Case对Test Fixture所做的任何更改都不会影响其它Test Case。

```
// 本代码中的test fixture
class TigerTest : public ::testing::Test
{
```

```
protected:
    void SetUp() override
    {
        cout << "TigerTest Test Case SetUp" << endl;
    }
    void TearDown() override
    {
        cout << "TigerTest Test Case TearDown" << endl;
    }

    Tiger tiger;
};

TEST_F(TigerTest, TrueName)
{
    EXPECT_EQ("Tiger", tiger.Name());
}

TEST_F(TigerTest, FalseName)
{
    EXPECT_EQ("Cat", tiger.Name());
}
```

四、gtest的main函数

使用以下的模板来开启并运行全部测试。

```
#include"gtest/gtest.h"
GTEST_API_ int main(int argc, char **argv) {
    testing::InitGoogleTest(&argc, argv);//解析命令行中的GoogleTest参数，它允许用户通过多样的命令行参数来控制测试程序的行为（即定制命令行参数的行为）
    return RUN_ALL_TESTS();//将会搜索不同的Test Case和不同的源文件中所有已经存在测试案例，然后运行它们，所有Test都成功时返回1，否则返回0。
}
```

五、启动本项目中的gtest

```
cd test
make
./gtest.out
```

4.3 Valgrind使用

4.3.1 实验目的

了解Valgrind，并使用其找出内存泄漏的代码。

4.3.2 实验内容与要求

独立完成实验内容，对实验步骤进行截图，并认真撰写实验报告。

要求

- 使用valgrind进行内存泄漏的检测

内容

一、认识valgrind

Valgrind是一套Linux下，开放源代码的仿真调试工具的集合。Valgrind由内核以及基于内核的其他调试工具组成。内核模拟了一个CPU环境，并提供服务给其他工具；而其他工具类似于插件，利用内核提供的服务完成各种特定的内存调试任务。本实验主要使用memcheck进行内存泄漏检查。

二、检测内存泄漏

本实验中部分动物类存在内存泄漏，执行`valgrind --tool=memcheck --leak-check=full ./bin/zoo_boom.out &> val.txt`启用memcheck的内存泄漏检查并开始分析。

打开val.txt文件，在该文件的末尾，我们看到

```
==7843== Process terminating with default action of signal 27 (SIGPROF)
==7843==    at 0x4B634FA: __open_nocancel (open64_nocancel.c:45)
==7843==    by 0x4B712DF: write_gmon (gmon.c:370)
==7843==    by 0x4B71B3E: _mcleanup (gmon.c:444)
==7843==    by 0x4A96FDD: __cxa_finalize (cxa_finalize.c:83)
==7843==    by 0x10A3E6: ??? (in /root/courses/SDCE/SDCE-exp4/bin/zoo_boom.out)
==7843==    by 0x4011F6A: _dl_fini (dl-fini.c:138)
==7843==    by 0x4A968A6: __run_exit_handlers (exit.c:108)
==7843==    by 0x4A96A5F: exit (exit.c:139)
==7843==    by 0x4A74089: (below main) (libc-start.c:342)
==7843==
==7843== HEAP SUMMARY:
==7843==     in use at exit: 1,075,338 bytes in 1,005 blocks
==7843==   total heap usage: 1,020 allocs, 15 frees, 2,199,746 bytes allocated
==7843==
==7843== 8 bytes in 1 blocks are definitely lost in loss record 1 of 6
==7843==    at 0x483BE63: operator new(unsigned long) (in /usr/lib/x86_64-linux-
gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==7843==    by 0x10BFB3: init() (main.cpp:13)
==7843==    by 0x10C09C: main (main.cpp:23)
==7843==
==7843== 8 bytes in 1 blocks are definitely lost in loss record 2 of 6
==7843==    at 0x483BE63: operator new(unsigned long) (in /usr/lib/x86_64-linux-
gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
```

```
==7843==    by 0x10BFE6: init() (main.cpp:15)
==7843==    by 0x10C09C: main (main.cpp:23)
==7843==
==7843== 4,000 bytes in 1,000 blocks are definitely lost in loss record 3 of 6
==7843==    at 0x483BE63: operator new(unsigned long) (in /usr/lib/x86_64-linux-
gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==7843==    by 0x10BED1: Mouse::Drink() (mouse.cpp:23)
==7843==    by 0x10C1E9: Animal::Live() (animal.h:16)
==7843==    by 0x10C10A: main (main.cpp:26)
==7843==
==7843== 1,048,608 (32 direct, 1,048,576 indirect) bytes in 1 blocks are definitely
lost in loss record 6 of 6
==7843==    at 0x483BE63: operator new(unsigned long) (in /usr/lib/x86_64-linux-
gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==7843==    by 0x10C019: init() (main.cpp:17)
==7843==    by 0x10C09C: main (main.cpp:23)
==7843==
==7843== LEAK SUMMARY:
==7843==    definitely lost: 4,048 bytes in 1,003 blocks
==7843==    indirectly lost: 1,048,576 bytes in 1 blocks
==7843==      possibly lost: 0 bytes in 0 blocks
==7843==    still reachable: 22,714 bytes in 1 blocks
==7843==         suppressed: 0 bytes in 0 blocks
==7843== Reachable blocks (those to which a pointer was found) are not shown.
==7843== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==7843==
==7843== For lists of detected and suppressed errors, rerun with: -s
==7843== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 0 from 0)
```

我们观察record和LEAK SUMMARY发现，`Mouse::Drink()` 和`init()`中的`new`存在着内存泄漏。在定位出这两个问题之后，我们可以尝试修复这个bug。

4.4 Gprof使用

4.4.1 实验目的

查看当前程序中的调用链并将其可视化，分析恶意占用CPU的代码。

4.4.2 实验内容与要求

独立完成实验内容，对实验步骤进行截图，并认真撰写实验报告。

要求

- 使用gprof进行函数调用链的分析。
- 使用gprof分析恶意占用CPU的代码。
- 将gprof的分析内容可视化。

内容

一、gprof介绍

gprof(GNU profiler)是GNU binutils工具集中的一个工具，linux系统当中会自带这个工具。它可以分析程序的性能，能给出函数调用时间、调用次数和调用关系，找出程序的瓶颈所在。在编译和链接选项中都加入-pg之后，gcc会在每个函数中插入代码片段，用于记录函数间的调用关系和调用次数，并采集函数的调用时间。

二、gprof使用及可视化

1. 用gcc、g++编译程序时，使用-pg参数。如：`g++ -pg -o test.exe test.cpp`（本项目中编译比较复杂，可直接make）

   编译器会自动在目标代码中插入用于性能测试的代码片断，这些代码在程序运行时采集并记录函数的调用关系和调用次数，并记录函数自身执行时间和被调用函数的执行时间。

2. 执行编译后的可执行程序，生成文件`gmon.out`。

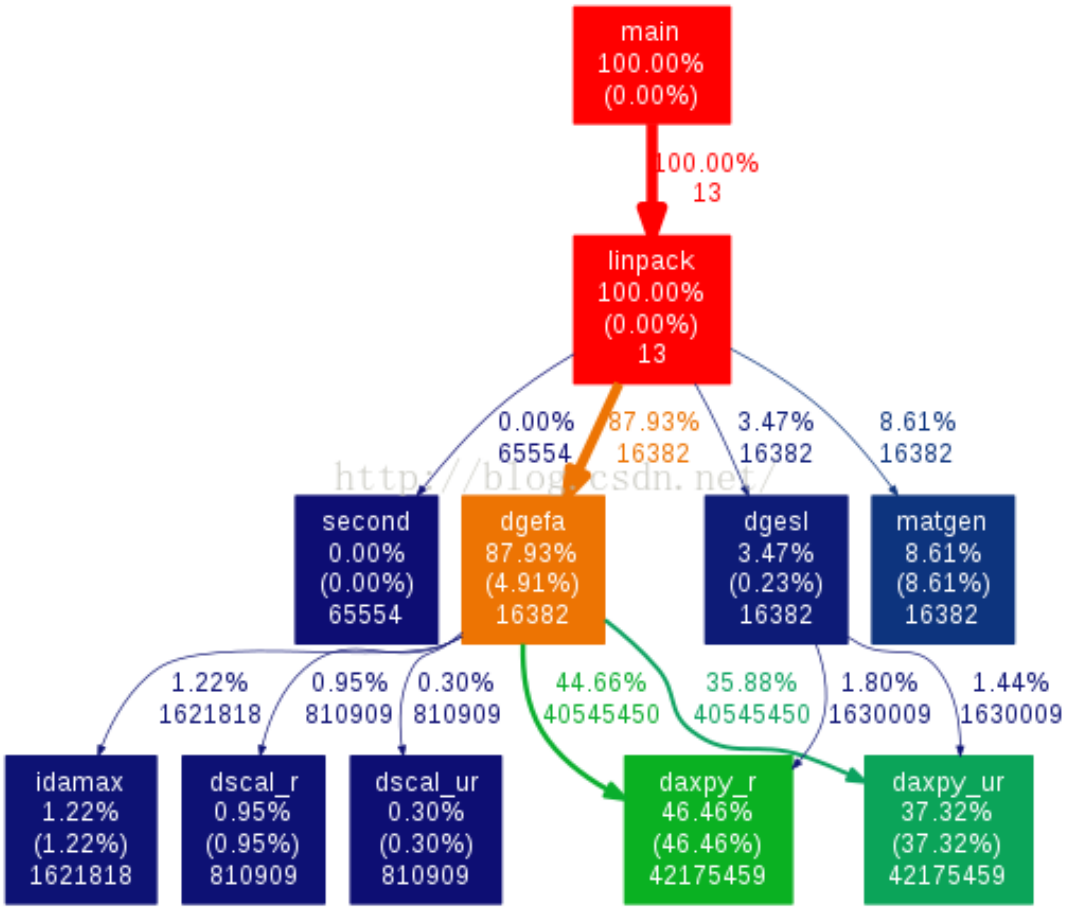
   该步骤运行程序的时间会稍慢于正常编译的可执行程序的运行时间。程序运行结束后，会在程序所在路径下生成一个缺省文件名为`gmon.out`的文件，这个文件就是记录程序运行的性能、调用关系、调用次数等信息的数据文件。

3. 使用gprof命令来分析记录程序运行信息的gmon.out文件。如：`gprof ./bin/zoo_boom.out gmon.out> gprofresult.txt`。该信息重定向到文本文件以便于后续分析。

4. 可视化：

   由于结果`gprofresult.txt`分析不太直观，可以借助gprof2dot.py与dot工具生成函数调用图

   - gprof2dot是一个Python脚本，用于将许多性能评测工具的输出转换为dot graph。其可以将perf, valgrind, python profilers, java HPROF等性能评测工具的输出转换为图片，如下：



- dot 是graphviz工具的一部分，其可以将 dot graph 转换为常用的图片格式。

可视化命令使用：

```
apt-get install graphviz # 安装dot
git clone https://github.com/jrfonseca/gprof2dot.git # 下载gprof2dot.py
chmod 755 gprof2dot.py # 修改gprof2dot项目中的gprof2dot.py的执行权限
gprof bch gmon.out |./gprof2dot.py -n0 -e0 |dot -Tpng -o output.png # 生成调用函数图
```

使用`gprof bin/zoo_boom.out gmon.out |./script/gprof2dot.py -n0 -e0 |dot -Tpng -o test/output.png`生成函数调用链图片`output.png`并保存到test文件夹中。

本项目中，以上的操作简化为

```
make clean
make
./bin/zoo_boom.out
make gprof
```

我们可以对`test/output.png`图片来进行调用链分析

三、分析恶意占用CPU的代码

在上一步中我们生成了gprofresult.txt文件。打开该文件，我们可以看到

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 54.21      0.13     0.13      100     1.30     1.30  Tiger::Eat()
 33.36      0.21     0.08  1048576     0.00     0.00  void
__gnu_cxx::new_allocator<BigMem>::construct<BigMem, BigMem>(BigMem*, BigMem&&)
  8.34      0.23     0.02      100     0.20     1.10  Wolf::Climb()
  4.17      0.24     0.01  1048576     0.00     0.00  void
std::allocator_traits<std::allocator<BigMem> >::construct<BigMem, BigMem>
(std::allocator<BigMem>&, BigMem*, BigMem&&)
```

该文件显示`Tiger::Eat()`函数中有占用了大量的CPU时间。我们进入该函数，发现这是一个无用的超大循环。注释掉该函数后，重新生成gprofresult，此时该函数的TIME指标下降，说明我们解决了恶意占用CPU的问题。

4.5 Perf使用

4.5.1 实验目的

使用perf进行CPU的分析并绘制火焰图。

4.5.2 实验内容与要求

独立完成实验内容，对实验步骤进行截图，并认真撰写实验报告。

要求

- 使用perf进行CPU使用的分析
- 使用perf画出火焰图

内容

一、perf介绍

perf 工具是 Linux 系统自带的一个性能分析工具，它可以用于分析应用程序、系统内核以及硬件的性能瓶颈，帮助开发者优化程序性能。perf 工具支持多种性能分析操作，包括 CPU 性能分析、内存分析、锁分析、网络分析等，可以提供详细的性能指标、函数调用图、热点分析等信息，帮助开发者诊断和解决性能问题。举例来说，使用 Perf 可以计算每个时钟周期内的指令数，称为 IPC，IPC 偏低表明代码没有很好地利用 CPU。Perf 还可以对程序进行函数级别的采样，从而了解程序的性能瓶颈究竟在哪里等等。Perf 还可以替代 strace，可以添加动态内核 probe 点，还可以做 benchmark 衡量调度器的好坏。

二、安装perf

perf的安装有以下方法

1. 通过包管理工具安装

```
# 下载linux-tools-common
$ sudo apt-get install linux-tools-common
# 查看是否存在perf
$ perf --version
# 如果不存在，可以下载特定的内核版本下的tools，根据命令行的提示，我的命令是， 这个命令需要下载特定内核版本的工具， 命令 uname -r 查看内核版本
$ sudo apt-get install linux-tools-5.4.0-122-generic
# 再检查一下
$ perf --version
perf version 5.4.192
```

2. 用源码安装perf

```
# 查看自己内核的版本，到官网上去下载特定的内核源码
>$ uname -r
5.4.0-122-generic
# 去官网上下载内核源码，可以手动下载，也可以使用wget
>$ wget http://ftp.sjtu.edu.cn/sites/ftp.kernel.org/pub/linux/kernel/v5.x/linux-5.4.122.tar.gz
# 下载完毕之后，接下内核源代码
>$ tar -zxvf linux-5.4.122.tar.gz
# 进入如下目录
>$ cd linux-5.4.122/tools/perf/
# 源码级安装，如有些依赖包没有安装，得安装一下，依赖包在下面第二个链接
```

```
>$ make -j10 && make install
# 查看perf的安装情况
>$ perf --version
perf version 5.4.192
```

3. wsl2中手动编译perf

```
apt install flex bison
git clone https://github.com/microsoft/WSL2-Linux-Kernel --depth 1
cd WSL2-Linux-Kernel/tools/perf
make -j8
sudo cp perf /usr/local/bin
```

三、使用stat来分析程序的指标

本实验中主要使用perf stat来进行程序的检测。stat于在运行指令时，监测并分析其统计结果。虽然perf top也可以指定pid，但是必须先启动应用才能查看信息。perf stat能完整统计应用整个生命周期的信息。

使用`perf stat ./bin/zoo_boom.out`来查看当前的程序：

```
 Performance counter stats for '/root/prog/part3/bin/zoo_boom.out':

            732.10 msec task-clock                #    1.000 CPUs utilized
                 0      context-switches          #    0.000 /sec
                 0      cpu-migrations            #    0.000 /sec
             6,783      page-faults               #    9.265 K/sec
     2,093,296,243      cycles                    #    2.859 GHz
     1,440,346,562      instructions              #    0.69  insn per cycle
       340,372,121      branches                  #  464.923 M/sec
           111,120      branch-misses             #    0.03% of all branches

       0.732469502 seconds time elapsed

       0.441484000 seconds user
       0.290978000 seconds sys
```

- task-clock(msec)是指程序运行期间占用了xx的任务时钟周期，该值高，说明程序的多数时间花费在 CPU 计算上而非 IO
- context-switches是指程序运行期间发生了xx次上下文切换，记录了程序运行过程中发生了多少次进程切换，频繁的进程切换是应该避免的。（有进程进程间频繁切换，或者内核态与用户态频繁切换）
- cpu-migrations 是指程序运行期间发生了xx次CPU迁移，即用户程序原本在一个CPU上运行，后来迁移到另一个CPU
- cycles：处理器时钟
- instructions: 机器指令数目，可通过cycles/instructions得出一条机器指令执行平均需要多少cycles
- 其他可以监控的指标譬如分支预测、cache命中、page-faults等

三、火焰图绘制

火焰图项目地址：git clone https://github.com/brendangregg/FlameGraph.git

1. 使用`perf record -F 999 -g ./bin/zoo_boom.out`来生成`perf.data`文件
2. 使用 perf script 工具对 perf.data 进行解析，生成折叠后的调用栈， 将解析出来的信息存下来, 供生成火焰图
   `perf script -i perf.data &> perf.unfold`
3. 再用 FlameGraph 中的 stackcollapse-perf.pl 将 perf 解析出的内容 perf.unfold 中的符号进行折叠，生成火焰图
   `./stackcollapse-perf.pl perf.unfold &> perf.folded`
4. 最后再用 FlameGraph 中的 flamegraph.pl 生成 svg 图：

   `./flamegraph.pl perf.folded > perf.svg`
5. 我们可以使用管道将上面的流程简化为一条命令

   `perf script -i perf.data | FlameGraph/stackcollapse-perf.pl | FlameGraph/flamegraph.pl > perf.svg`

火焰图会出现在process.svg文件中。