《软件安全》实验报告

姓名：王众 学号：2313211

实验名称

程序插桩及 实验

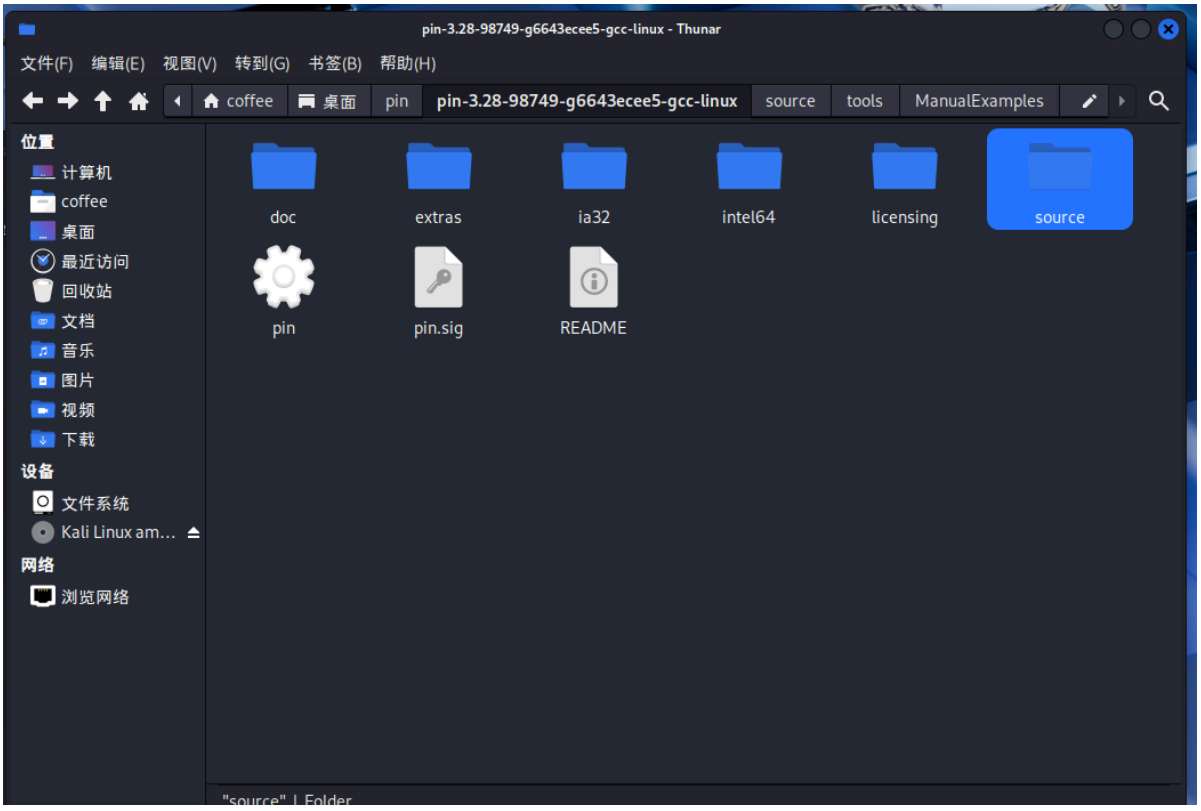
实验要求

复现实验一，基于$WindowsMyPinTool$或在$Kali$中复现$malloctrace$这个$PinTool$，理解Pin插桩工具的核心步骤和相关API，关注$malloc$和$free$函数的输入输出信息。

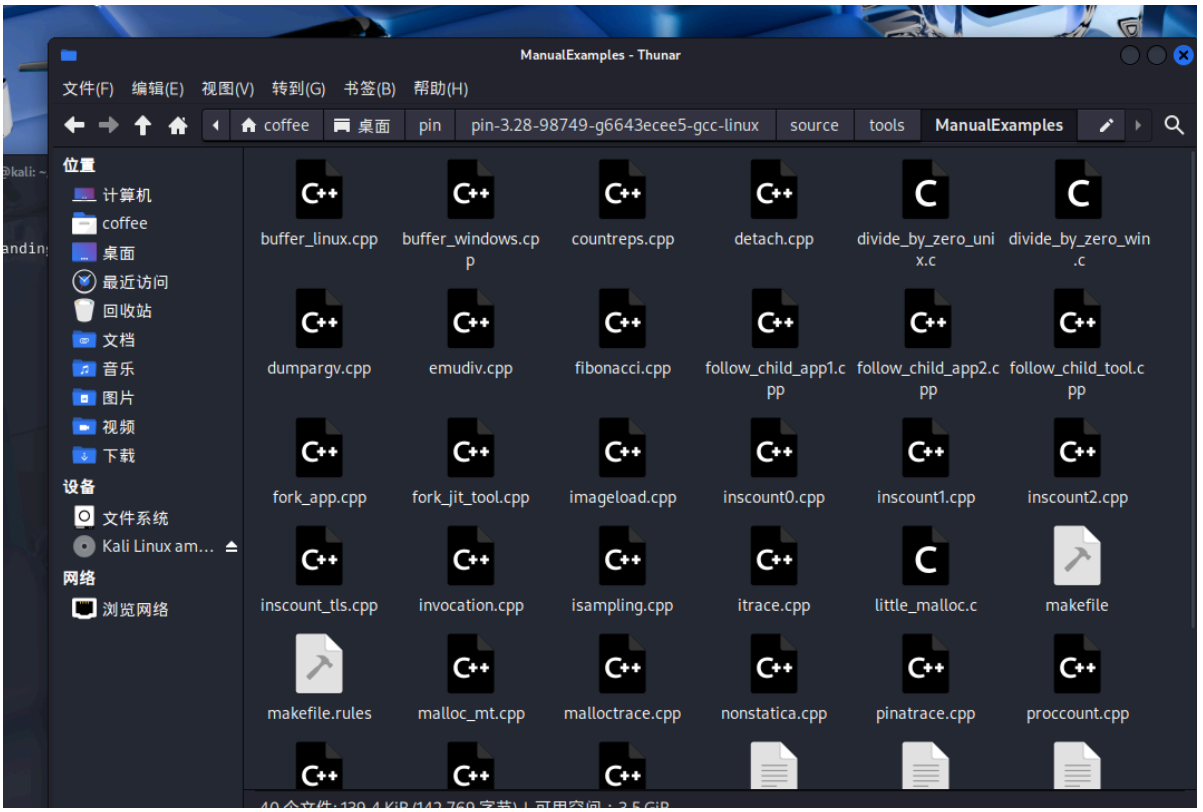
实验过程

1 在kali中安装Pin工具

首先，我们需要安装 Pin，选择了在 $Windows$系统下先进行下载，然后压缩包拖到$Linux$系统中。如图所示，我们选择最新版的进行安装即可，版本号为3.31 。然后我们将其拖入$kali$虚拟机， 并完成解压工作



我们按照给出的路径，打开文件夹中的$Source$中的$tool$的$ManualExamples$，就可以查看到很多已经编写好的$pintool$，如下所示：

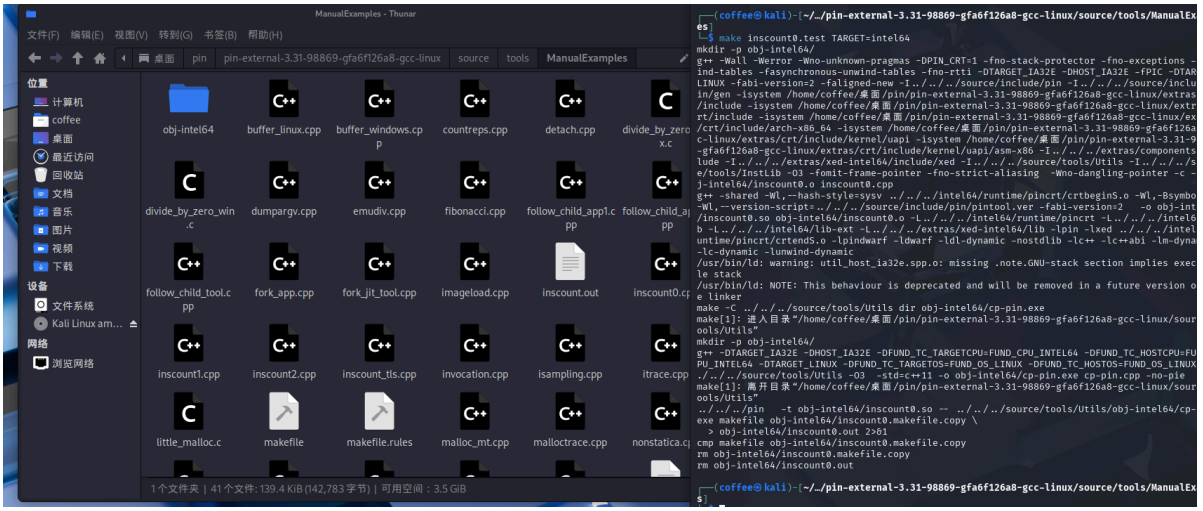


以上就是我们Pin的安装流程，接下来我们就可以开始实验了。

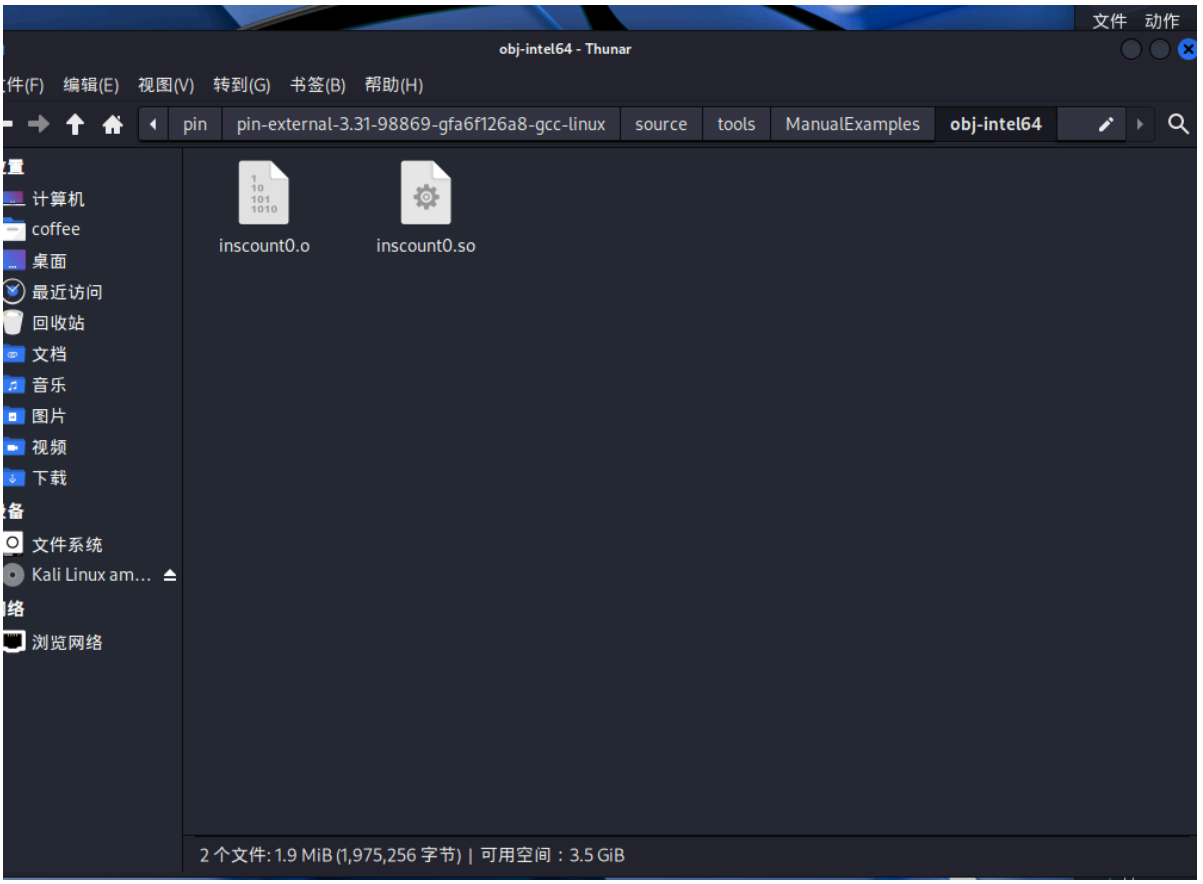
2 Pintool——malloctrace的使用

2.1 编译inscount0.cpp

我们首先打开文件夹 $ManualExamples$，选择文件`inscount0.cpp`，输入命令行`make inscount0.test TARGET=intel64`进行编译，输出如下所示：

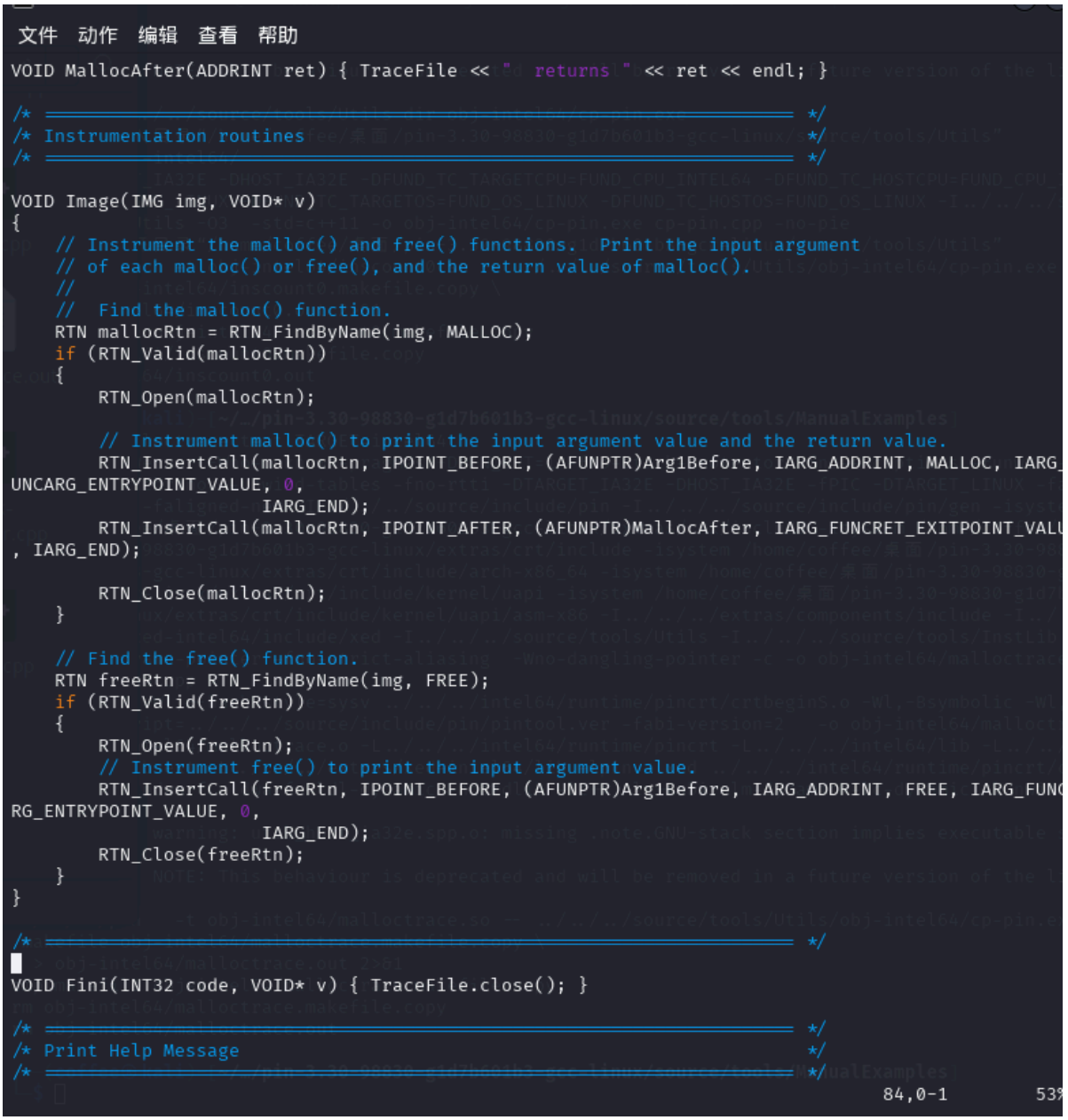


我们进入文件夹查看，发现文件夹中多出了一个已经完成编译的`inscount0.so`文件，说明文件编译完成！



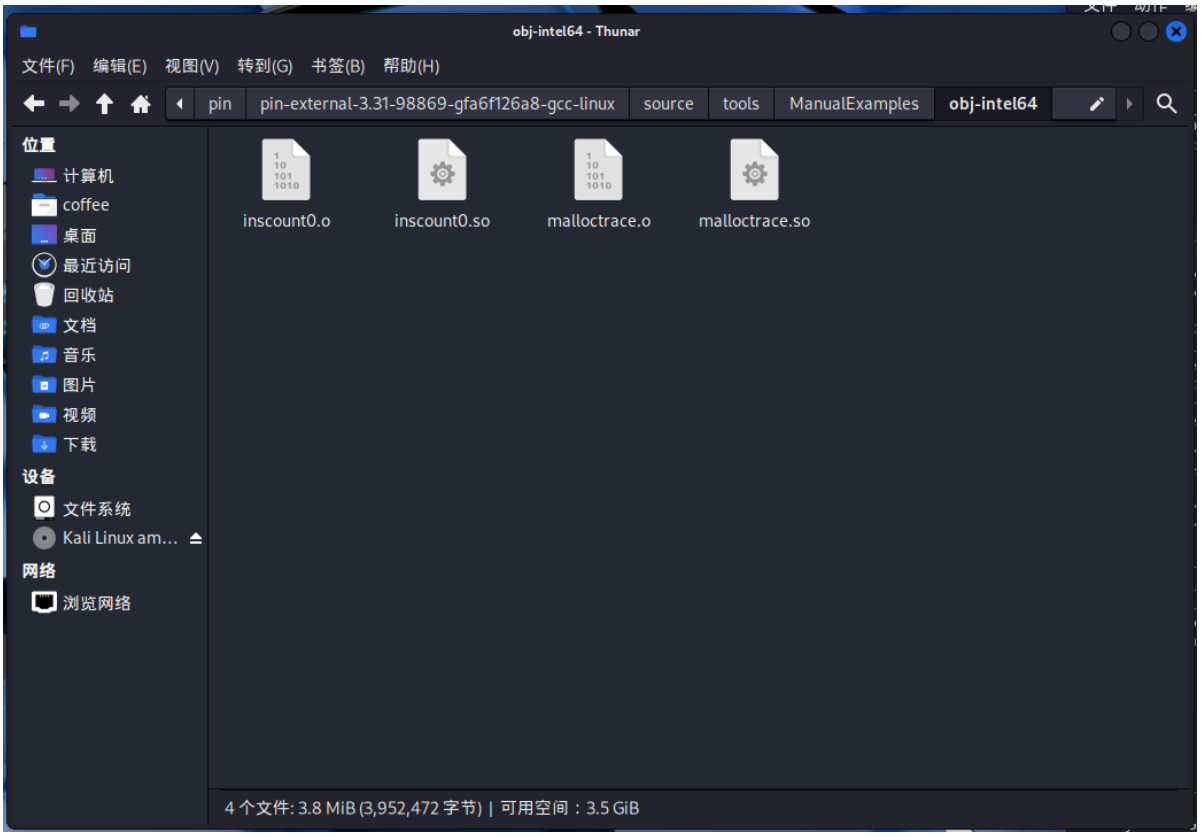
2.2 编译malloctrace.cpp产生动态链接库

我们按照实验要求，打开`source/tools/ManualExamples`，打开`malloctrace.cpp`进行查看源码：



```
文件  动作  编辑  查看  帮助
VOID MallocAfter(ADDRINT ret) { TraceFile << "  returns " << ret << endl; }

/* ===================================================================== */
/* Instrumentation routines                                              */
/* ===================================================================== */

VOID Image(IMG img, VOID* v)
{
    // Instrument the malloc() and free() functions.  Print the input argument
    // of each malloc() or free(), and the return value of malloc().
    //
    //  Find the malloc() function.
    RTN mallocRtn = RTN_FindByName(img, MALLOC);
    if (RTN_Valid(mallocRtn))
    {
        RTN_Open(mallocRtn);

        // Instrument malloc() to print the input argument value and the return value.
        RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)Arg1Before, IARG_ADDRINT, MALLOC, IARG_
UNCARG_ENTRYPOINT_VALUE, 0,
                       IARG_END);
        RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)MallocAfter, IARG_FUNCRET_EXITPOINT_VALU
, IARG_END);

        RTN_Close(mallocRtn);
    }

    // Find the free() function.
    RTN freeRtn = RTN_FindByName(img, FREE);
    if (RTN_Valid(freeRtn))
    {
        RTN_Open(freeRtn);
        // Instrument free() to print the input argument value.
        RTN_InsertCall(freeRtn, IPOINT_BEFORE, (AFUNPTR)Arg1Before, IARG_ADDRINT, FREE, IARG_FUNC
RG_ENTRYPOINT_VALUE, 0,
                       IARG_END);
        RTN_Close(freeRtn);
    }
}

/* ===================================================================== */
VOID Fini(INT32 code, VOID* v) { TraceFile.close(); }
/* ===================================================================== */
/* Print Help Message                                                    */
/* ===================================================================== */
                                                              84,0-1        53%
```

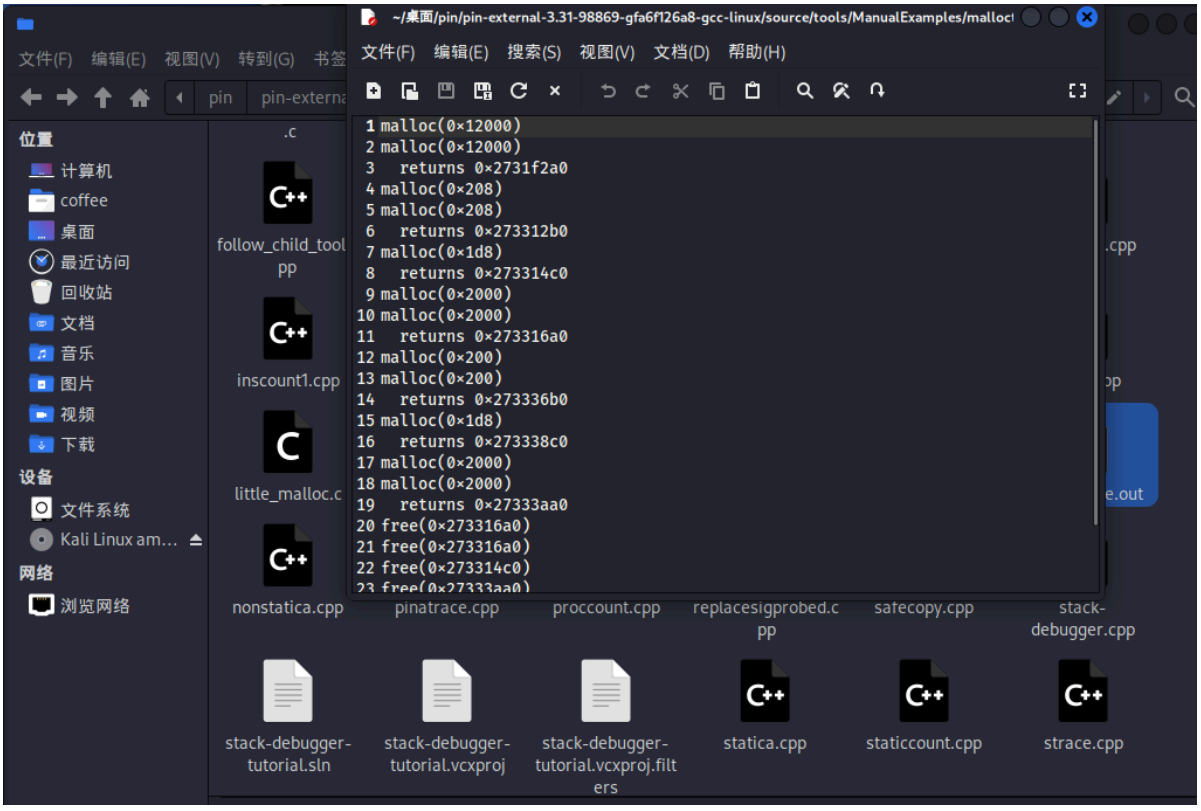
我们输入命令行 `make malloctrace.test TARGET=intel64` ，编译运行该文件，得到以下的结 果：



我们看到其中产生了 `malloctrace.so` 文件，即成功产生了动态链接库。

2.3 进行插桩实验

在进行完以上的步骤之后，我们就可以应用这个`tool`来进行插桩实验了！我们上一部分已经完成了对文件的编译，我们直接使用上面的编译生成的 `out` 文件进行查看，发现确实生成了一大串文字！我们打开文件，查看里面的内容，发现其输出了 `malloc` 和 `free` 函数的具体数值。



我们发现输出的文字由 `malloc` 函数、 `returns` 、 `free` 函数构成，我们在下一个部分我们具体进行 介绍。可以发现，每次使用 `malloc` 申请堆内存空间的时候，都会输出申请的空间的大小，并且得到申请 的空间的起始地址。当使用 `free` 释放内存空间的时候，会输出释放的空间的起始地址。

3 malloc和free函数输入输出的理解

我们根据上面的输出内容来进一步分析。

```
malloc(0x12000)
malloc(0x12000)
  returns 0x32aad2a0
malloc(0x208)
malloc(0x208)
  returns 0x32abf2b0
malloc(0x1d8)
  returns 0x32abf4c0
malloc(0x2000)
malloc(0x2000)
  returns 0x32abf6a0
malloc(0x200)
malloc(0x200)
  returns 0x32ac16b0
malloc(0x1d8)
  returns 0x32ac18c0
malloc(0x2000)
malloc(0x2000)
  returns 0x32ac1aa0
free(0x32abf6a0)
free(0x32abf6a0)
free(0x32abf4c0)
free(0x32ac1aa0)
free(0x32ac1aa0)
free(0x32ac18c0)
```

首先，前面的 `malloc` 都是系统读入程序后，进行的默认的内存空间的分配，括号中的内容应该是内 存空间的大小， `returns` 的值应该是内存进行存储的地址。然后，在使用完内存空间之后，程序会使用 `free` 函数进行释放，注意，释放的时候是读入内存，进行对应位置的内存释放即可。

4 Pintool基本框架

我们在此处分析一下 `Pintool` 的基本框架与源代码。

首先是 `malloctrace` 的分析。

```
#include "pin.H"
#include <iostream>
#include <fstream>
using std::cerr;
using std::endl;
using std::hex;
using std::ios;
```

```
using std::string;
std::ofstream TraceFile;
KNOB< string > KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o",
                              "malloctrace.out", "specify trace file name");
VOID Arg1Before(CHAR* name, ADDRINT size) { TraceFile << name << "(" << size <<
    ")" << endl; }
VOID MallocAfter(ADDRINT ret) { TraceFile << " returns " << ret << endl; }
VOID Image(IMG img, VOID* v)
{
    RTN mallocRtn = RTN_FindByName(img, MALLOC);
    if (RTN_Valid(mallocRtn))
    {
        RTN_Open(mallocRtn);
        RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)Arg1Before,
                       IARG_ADDRINT, MALLOC, IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                       IARG_END);
        RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)MallocAfter,
                       IARG_FUNCRET_EXITPOINT_VALUE, IARG_END);
        RTN_Close(mallocRtn);
    }
    RTN freeRtn = RTN_FindByName(img, FREE);
    if (RTN_Valid(freeRtn))
    {
        RTN_Open(freeRtn);
        RTN_InsertCall(freeRtn, IPOINT_BEFORE, (AFUNPTR)Arg1Before, IARG_ADDRINT,
                       FREE, IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                       IARG_END);
        RTN_Close(freeRtn);
    }
}
VOID Fini(INT32 code, VOID* v) { TraceFile.close(); }
INT32 Usage()
{
    cerr << "This tool produces a trace of calls to malloc." << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}
int main(int argc, char* argv[])
{
    // Initialize pin & symbol manager
    PIN_InitSymbols();
    if (PIN_Init(argc, argv))
    {
        return Usage();
    }
    // Write to a file since cout and cerr maybe closed by the application
    TraceFile.open(KnobOutputFile.Value().c_str());
    TraceFile << hex;
    TraceFile.setf(ios::showbase);
    // Register Image to be called to instrument functions.
    IMG_AddInstrumentFunction(Image, 0);
    PIN_AddFiniFunction(Fini, 0);
    // Never returns
    PIN_StartProgram();
    return 0;
}
```

两个分析函数是 `Arg1Before` 和 `MallocAfter`。`Arg1Before` 在调用`malloc`或`free` 之前执行，用于记录函数名称和参数。 `MallocAfter` 在调用 `malloc` 之后执行，记录返回值。 `Image`函数是用于在程序映像中找到并插桩`malloc` 和 `free` 函数的代码。它会在程序加载时被调用。对于找到的`malloc` 和`free`函数，代码使用 `RTN_InsertCall`在函数调用前后插入分析函数（如 `Arg1Before` 和 `MallocAfter`）。 `Fini`函数在 `Pin`工具完成后关闭输出文件。`Usage` 函数提供了一个帮助信息， 当命令行参数有误时显示。

以上是该程序的`Pintool`的基本框架，下面简单说明一下一般的 `Pintool`框架：

首先要进行初始化，通过调用函数 `PIN_Init` 实现 注册插桩函数。通过使用 `XXX_AddInstrumentFunction` 注册一个插桩函数，由于使用了指令级插 桩，因此在原始程序的每条指令执行前，都会进入到我们注册的插桩函数中，然后执行相应的操 作。

注册退出回调函数。通过使用`PIN_AddFiniFunctino`注册一个程序退出时的回调函数，当应用退出的时候会调用该函数。

启动程序。使用函数 `PIN_StartProgram` 启动程序。

心得体会：

通过学习，我深入了解了`Pin`工具的动态插桩技术及其在内存分配跟踪中的应用。实验通过`malloctrace.cpp`插桩`malloc`和`free`，记录参数和返回值，清晰展示了内存管理过程。我掌握了插桩点的使用和代码实现逻辑，认识到动态插桩在调试和性能分析中的强大功能。实验挑战提升了我的实践能力，激发了对系统编程的兴趣。未来，我希望探索更多插桩工具和内存优化方法，将所学应用于实际场景。