### Puzzle1: 基本块覆盖度插桩

### 查找插桩点的方法

LLVM pass 遍历模块中的每个函数，在每个函数内部，pass 遍历每个基本块，在每一个基本块代码之前插入函数 instrumentBasicBlocks 记录覆盖情况。

为了打印最终的基本块覆盖结果，在main 函数执行结束的位置插入函数 printBasicBlockCoverage 来打印最终的基本块覆盖结果。

### 插桩点进行插桩的手法

**在每个基本块的开始处，插入对外部函数 instrumentBasicBlocks 的调用**。基本思路是使用 getOrInsertFunction 获取外部函数，随后遍历所有函数中的基本块，在遍历中维护一个num值作为当前遍历到基本块的总数量。创建IRBuilder，使用IRBuildler中的SetInsertPoint方法指定插桩点为遍历到的基本块的开始位置，使用IRBuilder中的CreateCall插入外部函数并传入参数num代表该基本块的唯一编号。

**在main函数结束处，插入对外部函数 printBasicBlockCoverage的调用。**基本思路是通过 Function.getName() 判断函数的名称是否为main来判断当前函数是否为主函数，如果是则创建IRBuilder并使用SetInsertPoint指定插桩点为函数的末尾，使用CreateCall插入外部函数并传入参数num代表基本块的总数。

**代码实现部分如下：**

PreservedAnalyses run(Function &F, FunctionAnalysisManager &FAM)

    {

      FunctionCallee TestHello;

      FunctionCallee endShow;

      auto &M = \*F.getParent();

      AttributeList Attr;

      Attr = Attr.addFnAttribute(M.getContext(), Attribute::NoUnwind);

      IRBuilder<>

          IRB(M.getContext());

      // 查找符号名为printHello，返回值类型为void，参数列表类型为[void\*]的函数，注册到TestHello中

      TestHello = F.getParent()->getOrInsertFunction("instrumentBasicBlocks", Attr, IRB.getVoidTy(), IRB.getInt32Ty());

      endShow = F.getParent()->getOrInsertFunction("printBasicBlockCoverage", Attr, IRB.getVoidTy(), IRB.getInt32Ty());

      for (BasicBlock &BB : F)

      {

        num += 1;

        IRBuilder<> IRB(&BB);

        IRB.SetInsertPoint(&BB, BB.getFirstInsertionPt());

        IRB.CreateCall(TestHello, {IRB.getInt32(num)});

      }

      if (F.getName() == "main")

      {

        // Insert a call to "instrumentBasicBlocks" at the end of the function

        IRBuilder<> EndIRB(&F.back());

        EndIRB.SetInsertPoint(&F.back().back());

        EndIRB.CreateCall(endShow, {IRB.getInt32(num)});

      }

      return PreservedAnalyses::all();

    }

### 数据记录和输出的方法

在Myfunc.cpp中维护了全局集合CoveredBlocks记录基本块的执行情况，当一个基本块执行前，基本块执行代码前插入的instrumentBasicBlocks会向CoveredBlocks中存入当前基本块的唯一编号。

程序退出时调用printBasicBlockCoverage，调用时会传入当前代码基本块的总数量 num，最终使用 cout 输出覆盖基本块结果。

代码实现如下：

// MyFunc.cpp

#include "llvm/Support/raw\_ostream.h"

#include <unordered\_set>

#include <iostream>

using namespace std;

std::unordered\_set<int> CoveredBlocks;

extern "C" \_\_attribute\_\_((visibility("default"))) void instrumentBasicBlocks(uint32\_t BlockNumber) {

  CoveredBlocks.insert(BlockNumber);

}

extern "C" \_\_attribute\_\_((visibility("default"))) void printBasicBlockCoverage(uint32\_t allNumber) {

  cout << "Basic Block Coverage: " << CoveredBlocks.size() << "/" << allNumber << "\n";

}

### Puzzle2：共享变量识别

### 查找插桩点的方法

LLVM pass遍历模块中的每个函数，在每个函数内部，pass遍历每个基本块和基本块中的指令，使用动态类型检查（**dyn\_cast**）来确定指令是否为Load或Store指令,作为插桩点,在这两类指令处理时插入函数recordAccess。

为了打印最终的基本块覆盖结果，在main 函数执行结束的位置插入函数 reportSharedVariables来打印最终的结果。

### 插桩点进行插桩的手法

**在每个涉及全局变量的读/写指令处，插入对外部函数的recordAccess调用。**基本思路是使用 getOrInsertFunction 获取外部函数，然后创建 IRBuilder 并使用 SetInsertPoint 方法指定插桩点为当前的读/写指令位置，使用 CreateCall 插入外部函数调用，通过下面两行代码分别获取指令操作的内存地址以及指令操作的变量名称，将这两个数据传入到外部函数中。recordAccess函数调用负责记录每个变量的访问情况，包括访问的变量地址和线程标识。

// 获取操作地址

auto \*Address = Builder.CreateBitCast(LI->getPointerOperand(),

Type::getInt8PtrTy(\*C));

// 获取操作变量名称

auto \*VariableName = Builder.CreateGlobalStringPtr(LI->getPointerOperand()->getName());

**在main函数结束处，插入对外部函数 reportSharedVariables的调用。**基本思路是通过 Function.getName() 判断函数的名称是否为main来判断当前函数是否为主函数，如果是则创建IRBuilder并使用SetInsertPoint指定插桩点为函数的末尾，使用CreateCall插入外部函数并传入参数num代表基本块的总数。

**代码实现部分如下：**

struct SharedVariablePass : public PassInfoMixin<SharedVariablePass>

  {

    PreservedAnalyses run(Module &M, ModuleAnalysisManager &)

    {

      for (auto &F : M)

      {

        for (auto &B : F)

        {

          for (auto &I : B)

          {

            if (auto \*LoadInstruction = dyn\_cast<LoadInst>(&I))

            {

              instrumentLoad(LoadInstruction, &M.getContext());

            }

            else if (auto \*StoreInstruction = dyn\_cast<StoreInst>(&I))

            {

              instrumentStore(StoreInstruction, &M.getContext());

            }

          }

        }

        if (F.getName() == "main")

        {

          FunctionCallee TestHello;

          FunctionCallee endShow;

          auto &M = \*F.getParent();

          AttributeList Attr;

          Attr = Attr.addFnAttribute(M.getContext(), Attribute::NoUnwind);

          IRBuilder<>

              IRB(M.getContext());

          endShow = F.getParent()->getOrInsertFunction("reportSharedVariables", Attr, IRB.getVoidTy(), IRB.getVoidTy());

          // Insert a call to "instrumentBasicBlocks" at the end of the function

          IRBuilder<> EndIRB(&F.back());

          EndIRB.SetInsertPoint(&F.back().back());

          EndIRB.CreateCall(endShow, {});

        }

      }

      return PreservedAnalyses::all();

    }

  private:

    void instrumentLoad(LoadInst \*LI, LLVMContext \*C)

    {

      // 插桩load指令的逻辑

      IRBuilder<> Builder(LI);

      auto \*Address = Builder.CreateBitCast(LI->getPointerOperand(), Type::getInt8PtrTy(\*C));

      auto \*VariableName = Builder.CreateGlobalStringPtr(LI->getPointerOperand()->getName());

      std::vector<Value \*> Args = {Address, VariableName};

      FunctionCallee RecordAccessFunc = LI->getModule()->getOrInsertFunction("recordAccess", Type::getVoidTy(\*C), Type::getInt8PtrTy(\*C), Type::getInt8PtrTy(\*C));

      Builder.CreateCall(RecordAccessFunc, Args);

    }

    void instrumentStore(StoreInst \*SI, LLVMContext \*C)

    {

      // 插桩store指令的逻辑

      IRBuilder<> Builder(SI);

      auto \*Address = Builder.CreateBitCast(SI->getPointerOperand(), Type::getInt8PtrTy(\*C));

      auto \*VariableName = Builder.CreateGlobalStringPtr(SI->getPointerOperand()->getName());

      std::vector<Value \*> Args = {Address, VariableName};

      FunctionCallee RecordAccessFunc = SI->getModule()->getOrInsertFunction("recordAccess", Type::getVoidTy(\*C), Type::getInt8PtrTy(\*C), Type::getInt8PtrTy(\*C));

      Builder.CreateCall(RecordAccessFunc, Args);

    }

  };

### 数据记录和输出的方法

在 MyFunc.cpp 中维护了数据结构来记录每个全局变量的访问线程信息。每当一个变量被线程访问时，该线程的标识符和操作的内存地址被记录下来，若有地址被多个线程操作，则可以判定该地址存储的变量属于共享变量。

在程序结束时，调用 reportSharedVariables 函数来分析这些信息，并识别出被多个线程访问的共享变量。最终，通过标准输出（cout）以 "shared variables: [var0, var1, var2]" 的格式输出所有识别出的共享变量。

**实现代码：**

// MyFunc.cpp

#include <pthread.h>

#include <unordered\_map>

#include <set>

#include <string>

#include <iostream>

#include <algorithm>

using namespace std;

extern "C" {

// 全局数据结构，用于跟踪共享变量名称

std::unordered\_map<void\*, std::string> variableNames;

// 地址被哪些线程访问

std::unordered\_map<void\*, std::set<pthread\_t>> variableAccess;

std::set<std::string> sharedVariables;

void reportSharedVariables() {

    // 报告所有发现的共享变量

    std::cout << "shared variables: [";

    bool first = true;

    for (const auto& varName : sharedVariables) {

        if (!first) {

            std::cout << ", ";

        }

        std::cout << varName;

        first = false;

    }

    std::cout << "]" << std::endl;

}

void recordAccess(void\* variableAddress, const char\* variableName) {

    pthread\_t currentThread = pthread\_self();

    variableAccess[variableAddress].insert(currentThread);

    variableNames[variableAddress] = std::string(variableName);

    // 如果此变量被多于一个线程访问，则标记为共享

    if (variableAccess[variableAddress].size() > 1) {

        sharedVariables.insert(variableName);

    }

}

} // extern "C"

### puzzle3：lockset算法

### 查找插桩点的方法

LLVM pass遍历模块中的每个函数，在每个函数内部，pass遍历每个基本块及其内的指令。针对每个基本块中的指令，关注Load（加载）和Store（存储）指令，因为这些指令涉及对变量的读写操作，是数据竞争识别的关键点，在这两类指令处理时插入函数accessVariable。

此外，还关注函数调用指令，特别是涉及pthread互斥锁操作的函数调用（如pthread\_mutex\_lock和pthread\_mutex\_unlock），以跟踪线程间的锁操作，在这两个函数处理时分别插入函数lockVariable以及unlockVariable。

为了打印最终的基本块覆盖结果，在main 函数执行结束的位置插入函数 reportSharedVariables来打印最终的结果。

### 插桩点进行插桩的手法

**在每个Load或Store指令之前，插入对外部函数accessVariable的调用**。此函数接收当前操作的变量内存地址、变量名称以及操作类型（读/写）作为参数。这个函数的设计是为了记录每次对变量的访问，并在发现潜在的数据竞争时做出响应。

**对于互斥锁操作的函数调用指令，插入对lockVariable或unlockVariable的调用，以跟踪线程持有的锁**。插桩的实现方法是使用IRBuilder类在Load/Store指令或锁操作函数调用处创建新的函数调用指令,具体实现方法是通过判断函数名称是否为 pthread\_mutex\_lock 或是 pthread\_mutex\_unlock，如果是则插入函数lockVariable或是unlockVariable。同时通过instrument中的getArgOperand(0)获取到上锁（解锁）地址。

**在main函数结束处，插入对外部函数 reportRaceVariables的调用。**

**实现代码如下：**

// MyPass.cpp

#include "llvm/IR/PassManager.h"

#include "llvm/Passes/PassBuilder.h"

#include "llvm/Passes/PassPlugin.h"

#include "llvm/IR/IRBuilder.h"

#include "llvm/IR/Instructions.h"

#include "llvm/IR/Module.h"

#include "llvm/IR/Function.h"

using namespace llvm;

namespace

{

    struct RaceVariablePass : public PassInfoMixin<RaceVariablePass>

    {

        void instrumentLoad(LoadInst \*LI, LLVMContext \*C)

        {

            // 插桩load指令的逻辑

            IRBuilder<> Builder(LI);

            auto \*Address = Builder.CreateBitCast(LI->getPointerOperand(), Type::getInt8PtrTy(\*C));

            auto \*VariableName = Builder.CreateGlobalStringPtr(LI->getPointerOperand()->getName());

            std::vector<Value \*> Args = {Address, VariableName, Builder.getInt1(false)};

            FunctionCallee RecordAccessFunc = LI->getModule()->getOrInsertFunction("accessVariable", Type::getVoidTy(\*C), Type::getInt8PtrTy(\*C), Type::getInt8PtrTy(\*C));

            Builder.CreateCall(RecordAccessFunc, Args);

        }

        void instrumentStore(StoreInst \*SI, LLVMContext \*C)

        {

            // 插桩store指令的逻辑

            IRBuilder<> Builder(SI);

            auto \*Address = Builder.CreateBitCast(SI->getPointerOperand(), Type::getInt8PtrTy(\*C));

            auto \*VariableName = Builder.CreateGlobalStringPtr(SI->getPointerOperand()->getName());

            std::vector<Value \*> Args = {Address, VariableName, Builder.getInt1(true)};

            FunctionCallee RecordAccessFunc = SI->getModule()->getOrInsertFunction("accessVariable", Type::getVoidTy(\*C), Type::getInt8PtrTy(\*C), Type::getInt8PtrTy(\*C));

            Builder.CreateCall(RecordAccessFunc, Args);

        }

        void handleMutex(CallInst \*CI, LLVMContext \*C)

        {

            Function \*CalledFunction = CI->getCalledFunction();

            if (CalledFunction == nullptr || CalledFunction->getName().empty())

            {

                return;

            }

            // 检测函数是否为pthread的锁操作

            std::string FunctionName = CalledFunction->getName().str();

            if (FunctionName == "pthread\_mutex\_lock" || FunctionName == "pthread\_mutex\_unlock")

            {

                IRBuilder<> Builder(CI);

                auto \*LockAddress = CI->getArgOperand(0);

                Value \*Args[] = {LockAddress};

                std::string FuncName = (FunctionName == "pthread\_mutex\_lock") ? "lockVariable" : "unlockVariable";

                FunctionCallee MutexFunc = CI->getModule()->getOrInsertFunction(FuncName, Type::getVoidTy(\*C), Type::getInt8PtrTy(\*C));

                Builder.CreateCall(MutexFunc, Args);

            }

        }

        PreservedAnalyses run(Module &M, ModuleAnalysisManager &)

        {

            for (auto &F : M)

            {

                for (auto &B : F)

                {

                    for (auto &I : B)

                    {

                        if (auto \*LoadInstruction = dyn\_cast<LoadInst>(&I))

                        {

                            instrumentLoad(LoadInstruction, &M.getContext());

                        }

                        else if (auto \*StoreInstruction = dyn\_cast<StoreInst>(&I))

                        {

                            instrumentStore(StoreInstruction, &M.getContext());

                        }

                        else if (auto \*CallInstruction = dyn\_cast<CallInst>(&I))

                        {

                            handleMutex(CallInstruction, &M.getContext());

                        }

                    }

                }

                if (F.getName() == "main")

                {

                    FunctionCallee TestHello;

                    FunctionCallee endShow;

                    auto &M = \*F.getParent();

                    AttributeList Attr;

                    Attr = Attr.addFnAttribute(M.getContext(), Attribute::NoUnwind);

                    IRBuilder<>

                        IRB(M.getContext());

                    // 查找符号名为printHello，返回值类型为void，参数列表类型为[void\*]的函数，注册到TestHello中

                    endShow = F.getParent()->getOrInsertFunction("reportRaceVariables", Attr, IRB.getVoidTy(), IRB.getVoidTy());

                    // Insert a call to "instrumentBasicBlocks" at the end of the function

                    IRBuilder<> EndIRB(&F.back());

                    EndIRB.SetInsertPoint(&F.back().back());

                    EndIRB.CreateCall(endShow, {});

                }

            }

            return PreservedAnalyses::all();

        }

    };

} // end anonymous namespace

PassPluginLibraryInfo getPassPluginInfo()

{

    const auto Callback = [](PassBuilder &PB)

    {

        PB.registerPipelineStartEPCallback(

            [&](ModulePassManager &MPM, auto)

            {

                MPM.addPass(RaceVariablePass());

            });

    };

    return {LLVM\_PLUGIN\_API\_VERSION, "RaceVariablePass", "1.0", Callback};

}

extern "C" LLVM\_ATTRIBUTE\_WEAK PassPluginLibraryInfo llvmGetPassPluginInfo()

{

    return getPassPluginInfo();

}

### 数据记录和输出的方法

为实现功能定义了以下全局数据结构：

* **variableNames**：维护内存地址到变量名称的映射。这对于识别哪个变量参与了潜在的数据竞争至关重要。
* **locksets**：记录每个变量地址所关联的锁集合。这有助于确定是否存在任何未加锁的对共享变量的访问。
* **threadLocks**：记录每个线程当前持有的锁。这是理解线程如何同步对共享资源访问的关键。
* **visited**：记录每个变量地址第一次被哪个线程访问。这有助于识别变量是否被不持有锁的多个线程访问。
* **raceVariables**：存储识别出的所有存在数据竞争的变量名。

在**accessVariable**函数中，当一个线程访问变量时，首先记录变量地址和线程信息。检查当前访问是否可能引起数据竞争：

当一个变量地址第一次出现（即在**locksets**中没有找到）时，检查该地址是否已经被当前线程访问（在**visited**中记录）。如果是初次访问或者之前也是由同一线程访问的，更新**visited**记录，并将该变量地址与当前线程的锁集合关联（**locksets[variableAddress] = threadLocks[currentThread];**）。如果之前已经有不同线程访问过，且没有使用锁，那么将这个变量标记为涉及数据竞争（加入**raceVariables**）。

如果这个变量地址之前已经被访问过（在locksets中找到）：获取当前线程持有的锁集合（locks）和该变量地址已知的锁集合（addressLocks），计算两个锁集合的交集。如果交集为空，意味着至少有一个线程在没有变量对应锁的情况下访问了该变量，存在数据竞争，因此将变量加入到raceVariables，如果交集非空，说明所有访问这个变量的线程都至少持有一个共同的锁，将locksets中该地址的锁集合更新为交集，以维护最小的锁集合，保证对变量的后续访问仍然能够被正确检测。

**lockVariable** 与 **unlockVariable** 则需要传入pthread\_mutex\_t锁变量，使用pthread\_selft()函数获取当前线程唯一标识符，操作threadLocks添加或是移除传入的锁变量代表线程持有或是释放锁。

**reportRaceVariables()**函数中遍历raceVariables中的所有变量并输出最终的结果。