LLVM Pass for `printf()` Call Logging

Project Specification

The aim of this project is to write an LLVM Pass that transforms any given program that contains direct calls to **`printf()`**, so that all output to **`stdout`** via **`printf()`** is also logged in a file named **`log.txt`** using **`fprintf()`** right after each **`printf()`** call.
The file **`log.txt`** must be opened by **`fopen()`** and closed by **`fclose()`** at the entry and exit points of **`main()`**, accordingly.

Study Material

Everything useful required for the project was found either in the LLVM documentation or the LLVM mailing list. In particular, I used the following sources (more or less, in this order):

- **An Example Using the LLVM Toolchain:**
  https://llvm.org/docs/GettingStarted.html#an-example-using-the-llvm-tool-chain
- **Developing LLVM Passes out of Source:**
  https://llvm.org/docs/CMake.html#developing-llvm-passes-out-of-source
- **Writing an LLVM Pass:**
  https://llvm.org/docs/WritingAnLLVMPass.html
- **Implementing a Language with LLVM:**
  https://llvm.org/docs/tutorial/#kaleidoscope-implementing-a-language-with-llvm
- **LLVM Programmer's Manual:**
  https://llvm.org/docs/ProgrammersManual.html
- **LLVM Language Reference:**
  https://llvm.org/docs/LangRef.html

Regarding the mailing list, I did not save any specific emails, although they had to do mainly with bugs encountered during development, e.g. errors about malformed LLVM IR, such as missing terminator instruction in a Basic Block after transformation by the pass.

Solution Design & Implementation

Choosing an LLVM pass type

I opted for **`llvm::ModulePass`**, since this pass type provides complete control over the entire program and allows the programmer to add global variables and constants and also look up any function and create additional function definitions.

Handling the logfile

The file **`log.txt`** is opened with **`fopen()`** at the start of **`main()`** and closed with **`fclose()`** right before every exit point of **`main()`**, i.e. before every **`return`** instruction, as requested.

Discovering calls to `printf()`

Finding all direct calls to **`printf()`** is achieved by traversing all LLVM IR instructions in the program and picking those **`call`** or **`invoke`** instructions that refer to **`printf()`**.

Injecting calls to `fprintf()`

In order to inject the logging instructions, two cases must be considered, distinguished by the type of instruction used to call **`printf()`**:

1. **`call` instruction**
   - Adding a call to **`fprintf()`** in this case is as simple as inserting a **`call`** to **`fprintf()`** right after the **`call`** to **`printf()`**.
2. **`invoke` instruction**
   - Due to **`invoke`** being a terminator instruction, it must reside at the end of a block, thus blindly adding a call to **`fprintf()`** right after it would lead to malformed IR. Instead, the invocation to **`printf()`** is replaced with a **`call`** to **`printf()`** and an **`invoke`** instruction for **`fprintf()`** is added right after it, at the end of the block.

Avoiding crashes

In order to prevent the introduction of crashes, I made sure that logging is disabled if **`fopen()`** fails by wrapping each injected call to **`fprintf()`** and **`fclose()`** in a conditional.

Evaluation

The pass seems to work successfully in all tested cases (see **`test/hello.cpp`**).

A sample transformation can be found in **`test/example.cpp`**, which is also included at the end of this report for convenience.

Limitations

Per project requirements:

- **Module has to contain `main()`:** The program is only transformed if it contains **`main()`**.
- **`printf()` calls replaced with `puts()` calls or similar are not handled:** Clang may replace calls to **`printf()`** with **`puts()`** or similar if optimizations are enabled. However, since the project required handling only direct calls to **`printf()`**, those are ignored.

Per chosen solution:

- **`printf()` calls within inline `asm()` blocks are not considered:** Inline **`asm()`** blocks are transferred unchanged from C/C++ to LLVM IR, so inspecting and transforming them would require deep Assembly knowledge for every platform supported by LLVM and thus I decided that they constitute a special case, outside of the scope of this project.

Sample Transformation

Original source code

```cpp
#include <cstdio>

int main() {
    printf("Hi!\n");
    return 0;
}
```

Source code after transformation

Notice, that the actual transformation takes place in LLVM IR, however the transformed sample code is translated to C++ for easier understanding:

```cpp
#include <cstdio>

void* logfile = nullptr;
const char* logfilePath = "log.txt";
const char* fopenMode = "w+";
const char* fopenFailureMessage = "Failed to open ‘log.txt'. "
                                  "Logging of printf() calls disabled.\n";

void __LLVMPassForPrintfCallLogging_openLogfile() {
    if (logfile = fopen(logfilePath, fopenMode); logfile == nullptr) {
        printf(fopenFailureMessage);
    }
}

void __LLVMPassForPrintfCallLogging_closeLogfile() {
    if (logfile ≠ nullptr) {
        fclose(logfile);
    }
}

int main() {
    __LLVMPassForPrintfCallLogging_openLogfile();

    printf("Hi!\n");
    if (logfile ≠ nullptr) {
        fprintf(logfile, "Hi!\n");
    }

    __LLVMPassForPrintfCallLogging_closeLogfile();
    return 0;
}
```