Linking Lab: Reading Linkage from JCF Files

COMP 222: Introduction to Computer Organization
Assigned: 11/1/24, **Due: 11/15/24, 11:55 PM**

Important: This assignment must be done individually. Be sure to carefully read the course policies for assignments (including the honor code policy) on the assignments page of the course web site:

`http://www.clear.rice.edu/comp321/html/assignments.html`

Overview

You will complete the implementation of the `readjcf` program, which reads the linkage information from a Java Class File (JCF). The goals of this assignment are as follows:

- Gain an understanding of the similarities and differences between x86-64 ELF files and Java virtual machine class files.
- Improve your understanding of data representation and I/O in C.

Introduction

The compilation and execution of Java programs has both similarities and differences with respect to C programs. First, the `javac` compiler does not output x86-64 ELF files. It outputs files that are instead in the Java Class File (JCF) format. Second, the machine code within these files is not x86-64 machine code. It is machine code for a hypothetical computer called the Java Virtual Machine (JVM). Essentially, on CLEAR, this machine code is executed by a very fast JVM simulator[1]. Nonetheless, there are many conceptual similarities between the JCF and ELF file formats. In other words, to link together a Java program consisting of many classes, the JCF format must support many of the same features as the ELF file format. We learn about these similarities and differences through the implementation of the `readjcf` program.

First, we will explain the entire JCF format, then we will give a specification of the `readjcf` program that you will write. Although we need to describe the entire JCF format, the `constant_pool`, `fields`, and `methods` arrays are of particular interest for implementing the `readjcf` program.

Java Class File (JCF) Format:

The Java Class File (JCF) format is defined by the following pseudo-C structure. Note that in this definition, `u1`, `u2`, and `u4` refer to an unsigned one-, two-, and four-byte quantity, respectively, and that all multibyte data is stored in big-endian format:[2]

[1] This simulator is itself written in C++ and compiled to an x86-64 ELF executable.

[2] The full specification for the JCF format is located at `https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html`. This document presents the subset of the specification that is relevant to the assignment.

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

This structure is pseudo-C because the `constant_pool`, `interfaces`, `fields`, `methods`, and `attributes` arrays are variable sized, which is not allowed in C structures.

The `magic` field of a valid JCF must equal `0xCAFEBABE`. The `minor_version` and `major_version` fields contain version information about the class.

The `constant_pool` section of the JCF is most similar to a combination of the `.rodata` section, symbol table, and relocation information of an ELF file. The `constant_pool` contains all of the class' constants, including strings, numbers, references, and the names and types of classes, interfaces, fields, and methods. The `constant_pool` array contains one less entry than the count because the zeroth entry of the array is omitted. In other words, the first constant in the pool has index one.

The `access_flags` field is a bit vector that specifies access permissions for the class. The `this_class` and `super_class` fields are indices into the constant pool that reference the names of the current class and the super class. The `interfaces` section contains an array of indices into the constant pool for all of the interfaces that the class implements. The `fields` and `methods` arrays contain information about every single field and method in the class. Lastly, the `attributes` array contains specific attributes about the class.

There are fourteen different types of constants in the constant pool: `CONSTANT_Class`, `CONSTANT_Fieldref`, `CONSTANT_Methodref`, `CONSTANT_InterfaceMethodref`, `CONSTANT_String`, `CONSTANT_Integer`, `CONSTANT_Float`, `CONSTANT_Long`, `CONSTANT_Double`, `CONSTANT_NameAndType`, `CONSTANT_Utf8`, `CONSTANT_MethodHandle`, `CONSTANT_MethodType`, and `CONSTANT_InvokeDynamic`. The `cp_info` structures for all of these different constants all share the same general form:

```
cp_info {
    u1 tag;
    u1 info[];
}
```

The tag field indicates which type of constant the structure represents and, subsequently, the exact size of the `info` array.

The CONSTANT_Class constant contains the name of a class and is represented with the following structure:

```
CONSTANT_Class_info {
    u1 tag;
    u2 name_index;
}
```

The name_index field is a valid index into the constant_pool array. The element at that index corresponds to a CONSTANT_Utf8.

The CONSTANT_Fieldref, CONSTANT_Methodref, and CONSTANT_InterfaceMethodref constants indicate references made in the class to fields, methods, and interface methods, respectively. The following structure is used to represent all of these constants:

```
CONSTANT_{Field, Method, InterfaceMethod}ref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

The class_index and name_and_type_index fields indicate the class, name, and type of the field, method, or interface method being referenced. The fields are indices into the constant pool that correspond to CONSTANT_Class and CONSTANT_NameAndType constants, respectively.

The CONSTANT_String, CONSTANT_Integer, CONSTANT_Float, CONSTANT_Long, CONSTANT_Double, CONSTANT_MethodHandle, CONSTANT_MethodType, and CONSTANT_InvokeDynamic constants have the following structure definitions:

```
CONSTANT_String_info {
    u1 tag;
    u2 string_index;
}

CONSTANT_{Integer, Float}_info {
    u1 tag;
    u4 bytes;
}

CONSTANT_{Long, Double}_info {
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}

CONSTANT_MethodHandle_info {
    u1 tag;
    u1 reference_kind;
    u2 reference_index;
}
```

```
CONSTANT_MethodType_info {
    u1 tag;
    u2 descriptor_index;
}

CONSTANT_InvokeDynamic_info {
    u1 tag;
    u2 bootstrap_method_attr_info;
    u2 name_and_type_index;
}
```

Although you must necessarily pay attention to the sizes of the various data types in the above structures, the actual meaning of some of these structures' data members can remain opaque to you throughout the assignment. For instance, in the case of `CONSTANT_InvokeDynamic`, the `bootstrap_method_attr_info` is an index into the attributes table of the JCF. However, since you are not required to do any special processing of JCF attributes in your program, you may ignore what this actually refers to.

The JCF format specifies that the `CONSTANT_Long` and `CONSTANT_Double` constants occupy two entries in the constant pool. The specification says that, "If a `CONSTANT_Long_info` or `CONSTANT_Double_info` structure is the item in the `constant_pool` table at index n, then the next usable item in the pool is located at index $n + 2$. The `constant_pool` index $n + 1$ must be valid but is considered unusable."

The `CONSTANT_NameAndType` constant represents the name and type of a field or method. It is defined by the following structure:

```
CONSTANT_NameAndType_info {
    u1 tag;
    u2 name_index;
    u2 descriptor_index;
}
```

The `name_index` and `descriptor_index` fields indicate the name and description string of the field, method, or interface method being referenced. The fields are indices into the constant pool that correspond to `CONSTANT_Utf8` constants.

The `CONSTANT_Utf8` constant represents a UTF-8 string that is not NUL-terminated. UTF-8 is a variable-width character encoding that can represent any character in the Unicode character set, which includes over 110,000 characters from 100 scripts. UTF-8 is the most common character encoding used on the web, in part because ASCII encoding is a subset of UTF-8 character encoding. The `CONSTANT_Utf8` constant is defined by the following structure:

```
CONSTANT_Utf8_info {
    u1 tag;
    u2 length;
    u1 bytes[length];
}
```

The `field_info` and `method_info` structures both have the same definition:

```
{field, method}_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

The `access_flags` field is a bit vector that specifies access information about the field or method, including whether the method is public, protected, or private. The only flag that is necessary for this assignment is the `ACC_PUBLIC` flag. The `name_index` and `descriptor_index` are indices into the constant pool to a `CONSTANT_Utf8` that contains the name and description string of the field or method, respectively. Finally, the structure contains an array of attributes.

The `attribute_info` structure is defined by the following pseudo-C structure:

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

There are many more specific types of attributes defined by the Java Class File format, but they all follow the format of the previous structure. For this assignment, we will be ignoring all attribute information, so we do not need to delve into the specific types of attributes.

`readjcf` Specification:

The `readjcf` program will read a single Java Class File and print out the class' dependencies and exports, as requested by flags. Additionally, the program will perform some basic verification of the JCF. The program accepts three command line flags: "-d", "-e", and "-v". The flags "-d" and "-e" indicate whether or not the program should print the class file's dependencies and exports, respectively. The flag "-v" enables extra print statements for debugging. If no flags are specified, then no dependencies or exports should be printed, but the class file should still be read and verified.

We provide a main routine that reads and parses the command line arguments appropriately. The usage of the program is as follows:

```
readjcf [-d] [-e] [-v] <input file>
```

We will not use the `-v` flag when testing your program. Moreover, we are not specifying what the output from the extra print statements should be. That is up to you.

For the purposes of the `readjcf` program, we define a dependency as any reference made in the class, including self-references. These references are made by the `CONSTANT_Fieldref`, `CONSTANT_Methodref`, and `CONSTANT_InterfaceMethodref` constants in the constant pool. We define an export as any field or method in the class that has the access flag `ACC_PUBLIC` set.

The dependencies and exports should be printed in the order that they are encountered in the JCF. Due to the format of the JCF, this means that all of the dependencies will be printed before the exports. The output format for printing the dependencies and exports in the class file is exactly as follows:

```
Dependency - <class name>.<name> <descriptor>
Export - <name> <descriptor>
```

For a concrete example, the following is output from the reference solution:

```
Dependency - java/lang/Object.equals (Ljava/lang/Object;)Z
Dependency - java/lang/Object.hashCode ()I
Export - <init> (D)V
Export - toString ()Ljava/lang/String;
Export - equals (Ljava/lang/Object;)Z
Export - hashCode ()I
```

The program must also perform a simple form of verification of the JCF. Your `readjcf` program must verify the following:

- The magic number is `0xCAFEBABE`.
- The class file is not truncated and does not contain extra bytes.
- All indices into the constant pool that are followed while printing the dependencies and exports are valid and the constant at that index is of the expected type (according to its tag).

The verification should be performed as the file is processed. If at any time the file fails verification, the program should use the `readjcf_error` function to print an error message to `stderr`, then immediately quit.

We are providing you with a partially implemented program for reading these files. The provided program includes a main function that opens the input file and calls the rest of the processing functions in the correct order for the JCF format. You need to implement the following functions:

```
static int      print_jcf_constant(struct jcf_state *jcf,
                    uint16_t index, uint8_t expected_tag);
static int      process_jcf_header(struct jcf_state *jcf);
static int      process_jcf_constant_pool(struct jcf_state *jcf);
static void     destroy_jcf_constant_pool(struct jcf_constant_pool *pool);
static int      process_jcf_body(struct jcf_state *jcf);
static int      process_jcf_interfaces(struct jcf_state *jcf);
static int      process_jcf_attributes(struct jcf_state *jcf);
```

The `print_jcf_constant` function prints a constant from the constant pool to `stdout`. `process_jcf_header` reads the header (the `magic`, `minor_version`, and `major_version` fields) from the JCF. `process_jcf_constant_pool` reads and stores the constant pool, then prints out all of the class' dependencies, if requested. `destroy_jcf_constant_pool` deallocates all of the memory allocated to store the constant pool by `process_jcf_constant_pool`. `process_jcf_body` reads the body (the `access_flags`, `this_class`, and `super_class` fields) from the JCF. `process_jcf_interfaces` reads the interfaces count and the array of interfaces. `process_jcf_attributes` reads an attributes count, followed by an array of attributes.

Notes

- All of the multibyte data in the JCF is stored in big-endian format, which means that the data must be byteswapped before it may be accessed on CLEAR. To byteswap 16-bit and 32-bit integers use the functions `ntohs` and `ntohl`, respectively.
- The `jcf_cp_utf8_info` structure contains an array `bytes` of unspecified length as the last field of the structure. The compiler and the `sizeof()` operator do not include space for this array in the representation of the structure because it has an unknown number of elements. This causes the `bytes` field to occupy the bytes immediately after the structure. Thus, if you only allocate `sizeof(struct jcf_cp_utf8_info)` bytes, then the `bytes` field will occupy memory that is either not allocated or allocated for a different data object. In either case, this is problematic. If you instead allocate `(sizeof(struct jcf_cp_utf8_info) + numbytes)` bytes, you will have allocated memory for the `bytes` array. Specifically, the `bytes` array will be an array that is `numbytes` long.
- The UTF-8 strings in the constant pool are not NUL-terminated. The standard I/O library functions will work properly on NUL-terminated UTF-8 strings. You will need to NUL-terminate the strings yourself.
- The UTF-8 strings in the constant pool can be zero bytes long. The `fread` function will always return `0`, an indication of error, when it is asked to read an element whose `size` equals `0`.
- The `JCF_CONSTANT_Long` and `JCF_CONSTANT_Double` constants contain 8 bytes worth of data after the tag and occupy two spots in the constant pool. This means that, for example, you encounter one of these constants at index 1, then the next constant is located at index 3.
- The `access_flags` fields are bit vectors. Specific flags may be tested for by performing the boolean and of the field and the mask. For example, `access_flags & ACC_PUBLIC` will be true iff the `ACC_PUBLIC` flag is set.
- We provide you with the implementation of `process_jcf_fields` and `process_jcf_methods`. These functions provide a good example as to how to implement the rest of the program.
- The `fread` function can read structures from a file. This is the preferred way to read data in your program.

Getting Started

Please visit the web page at `https://classroom.github.com/a/4BcZdVx8`. (If you are copying this URL, do not include the period at the end of the sentence, as it is not part of the URL.) This page should say "RICE-COMP222-F24-Classroom" and "Accept the assignment — LinkingLab". Moreover, it should have a green button labeled "Accept this assignment"[3]. Please accept the assignment.

Upon accepting the assignment, you will be redirected to another web page. This page will confirm that you have accepted the assignment, and it will eventually (after you click refresh) provide you with a link to your personal repository for the assignment. Click this link to go to your personal repository.

The web page for your personal repository has a green button labeled "Code". Click this button. You should now see a text field with a URL. Copy or remember this URL.

Login to the `CLEAR` system if you have not already done so. Type the following:

[3] You may have to login to GitHub to see this page. If so, you will be prompted for your GitHub username and password.

`git clone` [Copy the URL for your repo here]

You will be prompted for your `github` username and password.

Once the clone operation is complete, you will have a directory named

`linkinglab-`[YOUR `github` ID]

Please `cd` into this directory, and run the command `ls`. You should see the following files:

- `readjcf.c` – provided code
- `readjcf_ref` – reference solution
- `Makefile` – specification for compiling `readjcf` using `make`
- `driver.pl` – compiles and runs your program on a set of test cases

If you do ***NOT*** see these files, contact the course staff immediately!

The executable `readjcf_ref` is the reference solution for `readjcf`. Run this program to resolve any questions you have about how your program should behave. *Your program should emit output that is byte-for-byte identical to the reference solution for any combination of the flags "-d" and "-e"*. You are not responsible for matching the exact output of the reference solution with the flag "-v". That said, attempting to match the output of the reference solution with the flag "-v" is a good way to develop and test your program in small steps.

Building Your Program

To build the program, use the Unix command:

```
make
```

This will compile your code and build the `readjcf` program. The `make` command will compile the `csapp.c` file and include it in your program.

Testing Your Program

To test your program, use the Unix command:

```
driver.pl
```

This will first use `make` to build your program, and then run it on the same test cases that we will use to grade your program.

Turning in Your Assignment

To turn in your code, you ***must*** use `git push` to copy your work to the `github` remote repository. We will *only* look at the last version that you `push`ed before the deadline. As a precaution against accidental loss of your code or writeup, we encourage you to `push` periodically. Please note, the *only* file that you need to turn in is `readjcf.c`. In other words, this is the only file on which you should ever perform `git add`.

For grading your submission, we will use the `Makefile` that was originally provided to compile your code. Therefore, your code should not rely on any modifications to the `Makefile` for correct compilation.

As a sanity check, you should use your web browser to visit your assignment repo. Make sure that what you see in the browser is consistent what you think you have pushed.