Lab 3 – Linker/Loader

Linker/Loader Programmer's Guide

CSE 3903

Spring 2023

Group: Worst Name Ever

Sade Ahmed

Jeremy Bogen

Mani Kamali

Giridhar Srikanth

Date of Submission: 04/19/2023

Table of Contents

Introduction

This is the programmer's guide to the WNE Linker Loader program. It first covers the program's data structures and data flow at a high level, then proceeds to go over low-level implementation details to make program modification easy and convenient. This guide assumes a familiarity with Java, its standard library, and Object Oriented Programming. For a guide on program usage, please refer to the WNE Linker Loader User's Guide.

Data Structures

The WNE linker loader is mainly built around two "data classes". These are classes that are only used to hold/modify data defined in their constructors, analogous to structs in other languages. These are used in the code to help clarify what certain methods return and what certain classes/methods take in as input.

PassOneResult

This data class defines the result of running through pass one. Here is its definition in the code:

```
static class PassOneResult {
     HashMap<String, Integer> symTable;
     int totalSize;
     ArrayList<Segment> segments;

     PassOneResult(HashMap<String, Integer> symTable, int
totalSize, ArrayList<Segment> segments) {
          this.symTable = symTable;
          this.totalSize = totalSize;
          this.segments = segments;
     }
}
```

It has three variables: **`symTable`**, **`totalSize`**, and **`segments`**. The **`symTable`** variable holds the external symbols defined throughout all the input files and their locations. As such, the **`symTable`** is what the second pass of the linker loader uses to populate the final object file. The **`totalSize`** variable simply holds the total combined size of the segments. And, finally, the

segments variable holds all the individual “segments”, as outlined in the Segment section of this guide.

Segment

The Segment data class holds all the information essential to processing a segment. Here is its code definition:

```
static class Segment {
     String segmentName;
     BufferedReader input;
     int size;
     int pla;
     HashMap<String, Integer> extSymbols;

     Segment(String segmentName, BufferedReader input, int size,
int pla, HashMap<String, Integer> extSymbols) {
               this.segmentName = segmentName;
               this.input = input;
               this.size = size;
               this.pla = pla;
               this.extSymbols = extSymbols;
     }
}
```

This class has five variables. The **segmentName** variable simply stores the name of the segment. The **input** variable holds the **BufferedReader** being used to read this particular segment’s file. The **size** variable holds the size of this segment. The **pla** variable holds the program load address of this particular segment. And, finally, the **extSymbols** variable is a map of all the external symbols defined in this segment.

Data Flow

High Level - The Main Method

The main method serves multiple important high-level data flow purposes. First, all user input is provided and parsed in the main method. Second, the main method coordinates calling pass one and pass two, managing any errors that pop up in either pass, and passing values between them. The following is the code corresponding to each bit.

First, the program creates an `ArrayList` of `BufferedReaders` for the input files. All arguments but the last are input files. So the code for this looks like so:

```
ArrayList<BufferedReader> inputs = new ArrayList<>();
System.out.print("Input files: ");
for (int i = 0; i < args.length - 1; i++) { // all but last value
(output file path)
        System.out.print(args[i] + " ");
        try {
            FileReader in = new FileReader(args[i]);
            inputs.add(new BufferedReader(in));
        }
        catch (IOException e) {
            System.err.println("Error: " + e.getMessage());
            cleanAndExit(inputs, -1);
        }
}
```

As you can see, the program starts by printing out each input file path. Then it attempts to open each path into a `BufferedReader`. If for whatever reason opening the files causes an IOException, that exception is printed and the program cleans up and exits with a `-1` error code.

Then, the program sets up the initial ipla value and gets the output file path:

```
int ipla = 0x3600;
String output = args[args.length - 1];
System.out.println("\nOutput file: " + output);
System.out.println("Running Pass One");
```

From here we enter the first pass's `try-catch` block. Each pass has its own try-catch block, any errors perpetuated from any methods in `PassOne.java` get passed up the chain till `main`, which will `catch` them here and report an error to the user, clean up any opened files, then exit.

This `try-catch` block starts with querying the user for an initial program load address (ipla):

```
try {
    Scanner scanner = new Scanner(System.in);
    ipla = Integer.parseInt(scanner.nextLine(), 16);
    passOne.ipla = ipla;
```

Then the program instatiates and executes `PassOne`:

```
     passOne.ipla = ipla;
     passOneResult = passOne.executePassOne(inputs);
```

And, then, finally checks to make sure the resulting program is all in one page.

```
     int ipla_after_page = (ipla + passOneResult.totalSize) >>> 9;
     int ipla_page = ipla >>> 9; // page is represented in the top 7
bits, by removing the bottom 9 we keep only what page it's in.
     if (ipla_page != ipla_after_page) {
                throw new RuntimeException("Program must fit in one
page, it currently does not.");
     }
     System.out.println("Pass one done");
```

From here, if no errors arise, then the program moves on to Pass Two. Which runs similarly:

```
    try {
           System.out.println("Running pass two");
           PassTwo passTwo = new PassTwo(passOneResult, output);
           passTwo.executePassTwo(output, ipla);
           System.out.println("Pass two done");
    } catch (Exception e) {
           System.err.println("Error: " + e.getMessage());
           cleanAndExit(inputs, -1);
    }
```

The program proceeds by printing "Linker done", cleaning up and exiting.

```
cleanAndExit(inputs, 0);
 System.out.println("Linker done");
```

Pass One

The PassOne class first defines the two Data Classes defined in PassOneResult and Segment. The `executePassOne` method goes through all the input segments and runs `processSegment`:

```
    for (BufferedReader input : inputs) {
        Segment seg = processSegment(input, pla);
        segments.add(seg);
        pla += seg.size;
        /* Example:
        ipla: 3600
        segment size: 3
        segment: 3600, 3601, 3602
        pla = 3600 + 3 = 3603
        segment size: 2
        segment: 3603, 3604
        etc.
        */
        extSymbolTable.putAll(seg.extSymbols);
    }
```

The processSegment method processes a single segment. It first gets the segment size from the header:

```
    HashMap<String, Integer> extSymbols = new HashMap<>();
    String line = input.readLine();
    String name = line.substring(1, 7); // H*Lib   *0003 (in hex)
    extSymbols.put(name, pla);
    int size = Integer.parseInt(line.substring(11), 16); // HLib
*0003* (in hex)
```

Then, `processSegment` will continually mark the Segment's `BufferedReader` and go through all of the file's N records until it hits a T record. It will then reset the BufferedReader, so that when `PassTwo` starts the `BufferedReader` points to the first T record.

```
     input.mark(300); // 300 bytes ahead, we're only doing one line so
this should be more than enough
     line = input.readLine();
     while (line.charAt(0) == 'N') { // Assume N records until first T
record
          int end = line.indexOf('=');
          String symbolName = line.substring(1, end);
          int symbolVal = Integer.parseInt(line.substring(end+1), 16)
+ pla;
          if (symbolVal > pla + size) {
               throw new RuntimeException("Symbol defined outside
segment: " + symbolName);
          }
          extSymbols.put(symbolName, symbolVal); // TODO this seems
fragile

          input.mark(300);
          line = input.readLine();
     }
     input.reset(); // Reset so the first line is a T record
     return new Segment(name, input, size, pla, extSymbols);
```

Execution then returns to `executePassOne`, which returns the previously defined `PassOneResult` object:

```
               extSymbolTable.putAll(seg.extSymbols);
     }
     int totalSize = pla - ipla;
     return new PassOneResult(extSymbolTable, totalSize,
segments);
```

Pass Two

The PassTwo class is used to hold the **PassOneResult**, output file path, and execute the linker's second pass. The constructor stores the first pass's result and opens the output file for reading:

```
PassTwo(PassOne.PassOneResult passOneResult, String
outputFilePath) throws IOException {
     this.passOneResult = passOneResult;
     this.writer = new BufferedWriter(new
FileWriter(outputFilePath));
}
```

The second pass only has one method: **executePassTwo.** This method starts by printing the output files Header record:

```
writer.write("HMain  " + String.format("%04X", ipla) +
String.format("%04X\n", passOneResult.totalSize));
```

The executePassTwo method then iterates over every line of every segment (remember that PassOne returns a BufferedReader pointing at the first T-record).

```
for (PassOne.Segment segment : passOneResult.segments) {
segment.input.lines().forEachOrdered((line) -> {
     …
```

Then, it will get the T-records location and value:

```
          int loc = Integer.parseInt(line.substring(1, 5), 16);
          loc = loc + segment.pla;
          short val = (short) Integer.parseInt(line.substring(5, 9),
16);
```

Then, if the T-record has an X9 relocation record, it will parse the value in the symbol table, 0 out the 7 bits, and reassign the **val** variable to the new relocated value. It resets the top 7 bits by doing **symVal &= 0x1FF;** This works because 0x1FF in binary is 0000 0001 1111 1111.

```
               if (line.length() > 9 && line.charAt(9) == 'X' &&
line.charAt(10) == '9') {
                    String sym = line.substring(11);
                    try {
```

```
                        short symVal = (short)
passOneResult.symTable.get(sym).intValue();
                        symVal &= 0x1FF;
                        val += symVal;
                    } catch(Exception e) {
                        throw new RuntimeException("Symbol \"" +
sym + "\" not defined");
                    }
            }
```

The logic for relocating X16 records is simpler, it simply rewrites the T-records value field:

```
            } else if (line.length() > 9 && line.charAt(9) == 'X' &&
line.charAt(10) == '1' && line.charAt(11) =='6') {
                    String sym = line.substring(12);
                    try {
                        val = (short)
passOneResult.symTable.get(sym).intValue();
                    } catch(Exception e) {
                        throw new RuntimeException("Symbol \"" +
sym + "\" not defined");
                    }
                }
```

And then writes out the new relocated T-record:

```
            try {
                    writer.write("T" + String.format("%04X", loc) +
String.format("%04x\n", val));
            } catch (IOException e) {
                    throw new RuntimeException(e);
            }
```

Then the second pass finishes by finding where the Main segment is in memory and assigning execution to start there, throwing an error if there's no Main segment:

```
    try {
            int main = passOneResult.symTable.get("Main  ");
            writer.write("E" + String.format("%04X", main));
            writer.flush();
```

```
      } catch (Exception e) {
                  throw new RuntimeException("Need a \"Main  \"
segment");
    }
```

And that concludes PassTwo.

App.java

```
/*
 * This Java source file was generated by the Gradle 'init' task.
 */
package lab3_wne_linker;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.Scanner;

public class App {
    public static void main(String[] args) {
        // Currently all args are inputs, so like run as ./gradlew
args="../examples/main.o ../examples/lib.o" and it'll output object file
to stdout
        ArrayList<BufferedReader> inputs = new ArrayList<>();
        System.out.print("Input files: ");
        for (int i = 0; i < args.length - 1; i++) { // all but last value
(output file path)
            System.out.print(args[i] + " ");
            try {
                FileReader in = new FileReader(args[i]);
                inputs.add(new BufferedReader(in));
            }
            catch (IOException e) {
                System.err.println("Error: " + e.getMessage());
                cleanAndExit(inputs, -1);
            }
        }

        int ipla = 0x3600; // java, fantastic language that it is, doesn't
support unsigned shorts, so we start with an int
        String output = args[args.length - 1];
```

```
        System.out.println("\nOutput file: " + output);
        System.out.println("Running Pass One");
        PassOne passOne = new PassOne(ipla);
        PassOne.PassOneResult passOneResult = null;
        try {
            System.out.println("Please enter the initial program load
address");
            Scanner scanner = new Scanner(System.in);
            ipla = Integer.parseInt(scanner.nextLine(), 16);
            passOne.ipla = ipla;
            passOneResult = passOne.executePassOne(inputs);
            int ipla_after_page = (ipla + passOneResult.totalSize) >>> 9;
            int ipla_page = ipla >>> 9; // page is represented in the top
7 bits, by removing the bottom 9 we keep only what page it's in.
            if (ipla_page != ipla_after_page) {
                throw new RuntimeException("Program must fit in one page,
it currently does not.");
            }
            System.out.println("Pass one done");
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
            cleanAndExit(inputs, -1);
        }

        try {
            System.out.println("Running pass two");
            PassTwo passTwo = new PassTwo(passOneResult, output);
            passTwo.executePassTwo(output, ipla);
            System.out.println("Pass two done");
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
            cleanAndExit(inputs, -1);
        }
        System.out.println("Linker done");
        cleanAndExit(inputs, 0);
    }

    static void printSymbolTable(HashMap<String, Integer> symTable) {
        symTable.forEach((sym, val) -> {
```

```
            System.out.println("sym: " + sym + " val: " +
Integer.toHexString(val));
        });
    }

    static void cleanAndExit(ArrayList<BufferedReader> inputs, int status)
{
        for (BufferedReader input : inputs) {
            try {
                input.close();
            } catch (Exception e) {
                System.err.println("Error: " + e.getMessage());
                System.exit(-1);
            }
        }
        System.exit(status);
    }
}
```

PassOne.java

```
package lab3_wne_linker;

import java.io.BufferedReader;
import java.io.IOException;
import java.lang.reflect.Array;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;

public class PassOne {

    int ipla;
    PassOne(int ipla) {
        this.ipla = ipla;
    }
```

```
    static class PassOneResult {
        HashMap<String, Integer> symTable;
        int totalSize;
        ArrayList<Segment> segments;

        PassOneResult(HashMap<String, Integer> symTable, int totalSize,
ArrayList<Segment> segments) {
            this.symTable = symTable;
            this.totalSize = totalSize;
            this.segments = segments;
        }
    }

    static class Segment {
        String segmentName;
        BufferedReader input;
        int size;
        int pla;
        HashMap<String, Integer> extSymbols;

        Segment(String segmentName, BufferedReader input, int size, int
pla, HashMap<String, Integer> extSymbols) {
            this.segmentName = segmentName;
            this.input = input;
            this.size = size;
            this.pla = pla;
            this.extSymbols = extSymbols;
        }
    }

    PassOneResult executePassOne(ArrayList<BufferedReader> inputs) throws
Exception {
        HashMap<String, Integer> extSymbolTable = new HashMap<>();
        ArrayList<Segment> segments = new ArrayList<>();
        int pla = ipla;
        for (BufferedReader input : inputs) {
            Segment seg = processSegment(input, pla);
            segments.add(seg);
            pla += seg.size;
            /* Example:
```

```
            ipla: 3600
            segment size: 3
            segment: 3600, 3601, 3602
            pla = 3600 + 3 = 3603
            segment size: 2
            segment: 3603, 3604
            etc.
            */
            extSymbolTable.putAll(seg.extSymbols);
        }
        int totalSize = pla - ipla;

        return new PassOneResult(extSymbolTable, totalSize, segments);
    }
    Segment processSegment(BufferedReader input, int pla) throws
IOException,RuntimeException {
        HashMap<String, Integer> extSymbols = new HashMap<>();
        String line = input.readLine();
        String name = line.substring(1, 7); // H*Lib   *0003 (in hex)
        extSymbols.put(name, pla);
        int size = Integer.parseInt(line.substring(11), 16); // HLib
*0003* (in hex)

        input.mark(300); // 300 bytes ahead, we're only doing one line so
this should be more than enough
        line = input.readLine();
        while (line.charAt(0) == 'N') { // Assume N records until first T
record
            int end = line.indexOf('=');
            String symbolName = line.substring(1, end);
            int symbolVal = Integer.parseInt(line.substring(end+1), 16) +
pla;
            if (symbolVal > pla + size) {
                throw new RuntimeException("Symbol defined outside
segment: " + symbolName);
            }
            extSymbols.put(symbolName, symbolVal); // TODO this seems
fragile

            input.mark(300);
```

```
            line = input.readLine();
        }
        input.reset(); // Reset so the first line is a T record
        return new Segment(name, input, size, pla, extSymbols);
    }
}
```

PassTwo.java

```
package lab3_wne_linker;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class PassTwo {
    PassOne.PassOneResult passOneResult;
    BufferedWriter writer;

    PassTwo(PassOne.PassOneResult passOneResult, String outputFilePath)
throws IOException {
        this.passOneResult = passOneResult;
        this.writer = new BufferedWriter(new FileWriter(outputFilePath));
    }

    void executePassTwo(String outputFile, int ipla) throws
RuntimeException, IOException {
        writer.write("HMain  " + String.format("%04X", ipla) +
String.format("%04X\n", passOneResult.totalSize));
        for (PassOne.Segment segment : passOneResult.segments) {
            segment.input.lines().forEachOrdered((line) -> {
                if (line.charAt(0) == 'T') {
                    //TODO: maybe clean up these parseInt calls into a
method
                    int loc = Integer.parseInt(line.substring(1, 5), 16);
                    loc = loc + segment.pla;
                    short val = (short) Integer.parseInt(line.substring(5,
9), 16);
```

```
                    if (line.length() > 9 && line.charAt(9) == 'X' &&
line.charAt(10) == '9') {
                        //val >>>= 9;
                        //val <<= 9; // in THEORY this resets the lower
nine bits to 0s (val >> 9) << 9
                        /*
                        Example:
                        1111 1111 1111 1111 >>> 9 = 0000 0000 0111 1111
                        0000 0000 0111 1111 << 9 = 1111 1110 0000 0000
                         */
                        String sym = line.substring(11);
                        try {
                            short symVal = (short)
passOneResult.symTable.get(sym).intValue();
                            symVal <<= 7;
                            symVal >>= 7; // get rid of top 7 bits
                            val += symVal;
                        } catch(Exception e) {
                            throw new RuntimeException("Symbol \"" + sym +
"\" not defined");
                        }
                    } else if (line.length() > 9 && line.charAt(9) == 'X'
&& line.charAt(10) == '1' && line.charAt(11) =='6') {
                        String sym = line.substring(12);
                        try {
                            val = (short)
passOneResult.symTable.get(sym).intValue();
                        } catch(Exception e) {
                            throw new RuntimeException("Symbol \"" + sym +
"\" not defined");
                        }
                    }
                    try {
                        writer.write("T" + String.format("%04X", loc) +
String.format("%04x\n", val));
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                }
            });
```

```
        }
        int main = passOneResult.symTable.get("Main  ");
        writer.write("E" + String.format("%04X", main));
        writer.flush();
    }
}
```