7. 程序汇编与链接

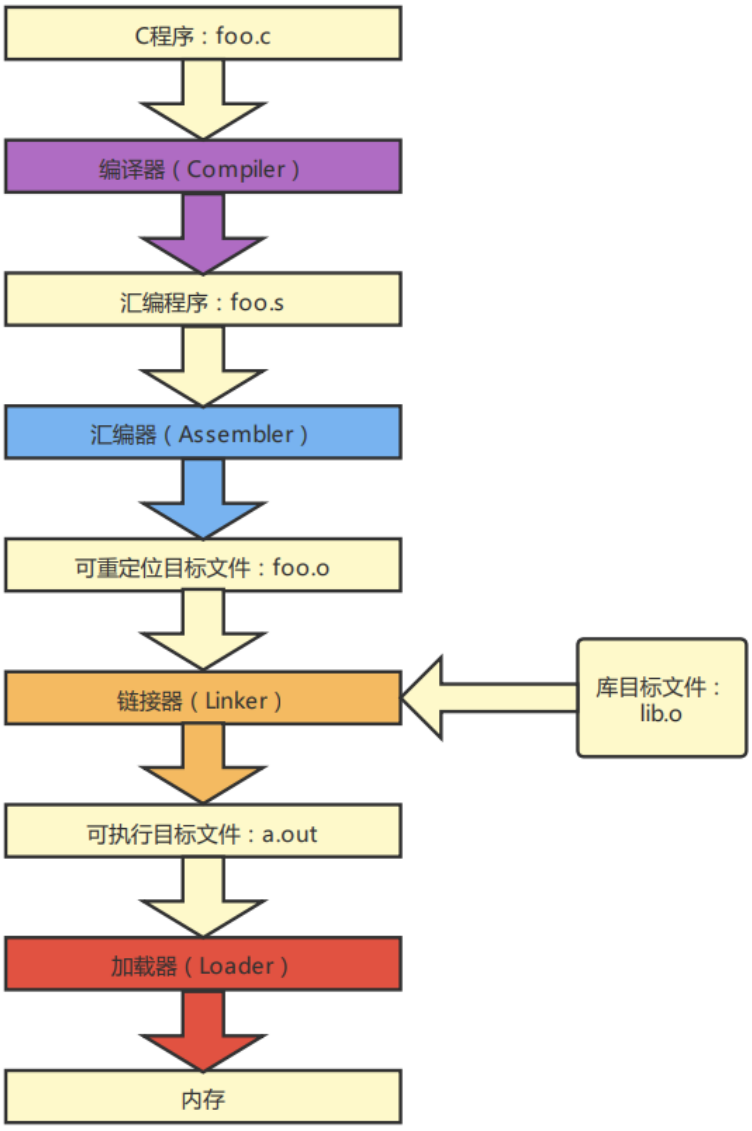
7.1 实验目的

1. 了解C语言程序的执行过程
2. 理解汇编器原理，了解如何从MIPS汇编文件进行符号解析和指令汇编。
3. 了解不同MIPS分支指令，相对地址和绝对地址的区别，掌握不同符号的地址解析。
4. 理解连接器原理，掌握如何利用汇编器生成的可重定位目标文件，构造符号表，关联不同文件中的符号，并进行重定位。

7.2 程序的启动

相信各位同学已经编写了不少高级语言的程序，我们可以理解自己编写代码的含义，但是计算机是如何理解并准确无误地执行我们的代码的。这一过程并不是一个启动按钮，或是一行指令。

实际上，程序的启动是一个相当复杂的过程，对于C语言而言，涉及了四个相对独立的功能模块，分别是：编译器（Compiler），汇编器（Assembler），链接器（Linker），加载器（Loader）。他们按照一定的次序工作，上一层的输出产物成为下一层的输入，逐步将程序转换成与实际计算机系统硬件相关的，计算机可以理解执行的程序（C.A.L.L.过程）。如下图所示：



- 为了确定文件类型UNIX使用文件的后缀，x.c表示C源文件，x.s表示汇编文件，x.o表示目标文件，x.a表示静态链接库，x.so表示动态链接库。默认情况下，a.out表示可执行文件。
- MS-DOS使用.C,.ASM,.OBJ,.LIB,.DLL,.EXE来完成相同功能。

7.2.1 代码编译

编译器将C程序转换成一种机器可以理解的符号形式的**汇编语言**程序，它能够被进一步翻译成二进制机器语言。

7.2.2 代码汇编

由编译器生成的汇编程序中，存在着简化程序转换和编程的一些实际硬件不支持的指令，这类指令称为**伪指令(pseudoinsruction)**。
例如，MIPS硬件确保寄存器`$zero`保持0。即一旦使用该寄存器，它都能提供0，而且程序员不能修改寄存器`$zero`的值。寄存器`$zero`用于生成汇编语言指令`move`。`move`的功能是将一个寄存器中的内容复制到另一个中。因此即使MIPS体系结构不存在这条指令，MIPS汇编器依然可以识别它。

```
1 move $t0, $t1 # 寄存器$t0得到寄存器$t1的值
```

汇编器将这条汇编语言指令转换成等价的机器语言指令：

```
1 add $t0, $zero, $t1
```

总的来说，伪指令使得MIPS拥有比硬件所实现的更为丰富的汇编语言指令集。唯一的代价是保留了`$at`寄存器。

汇编器的主要任务是汇编成机器代码。汇编器将汇编语言程序转换成**目标文件(object file)**，它包括机器语言指令，数据和指令正确放入内存所需要的信息。
为了产生汇编语言程序每条指令对应的二进制表示，汇编器必须处理所有标号对应的地址，汇编器将分支和数据传输指令中用到的标号都放入一个**符号表(Symbol Table)**中。这个表由标号和地址组成。
UNIX系统中的目标文件通常包括以下6个不同部分：

1. 目标文件头，描述目标文件其他部分的大小和位置
2. 代码段，包含机器语言代码
3. 静态数据段，包含在程序生命周期内分配的数据
4. 重定位信息，标记了一些在程序加载进内存时依赖于绝对地址的指令和数据
5. 符号表，包含未定义的剩余标记，如外部引用
6. 调试信息，包括了一份说明目标模块如何编译的简明描述

7.2.3 代码链接

到目前为止我们描述的所有内容表明，对源程序的任意一行代码的修改都需要重新编译和汇编整个程序，全部的编译和汇编效率是底下的，尤其对于标准库而言。因而出现了另一种方法，即单独编译和汇编么每个过程，使得某一行代码的修改只需要编译和汇编一个过程，这种方法需要一个新的系统程序，称为**链接器(Linker)**。它把所有独立汇编的机器语言程序“拼接”在一起。

链接器的工作分为三个步骤：

1. 将代码和数据模块象征性地放入内存
2. 决定数据和指令标签的地址
3. 修补内部和外部引用

链接器使用每个目标模块中的重定位信息和符号表，来解析所有未定义标签，这种引用发生在分支指令，跳转指令和数据寻址处。它的工作非常像一个编辑器，寻找所有旧地址并用新地址加以代替。
如果所有外部引用都解析完成，链接器接着要确定每个模块将要占用的内存位置。因为文件是单独汇编的，所以汇编器不可能知道该模块指令和数据相对于其他模块而言的位置，因此也就无从确定他们的内存位置。当链接器将一个模块放到内存中的时候，所有的**绝对引用**（与寄存器无关的内存地址），必须**重定位**来反映它们的真实地址。
链接器将产生一个**可执行文件(Executable file)**，它可以在一台计算机上运行，通常，这个文件与目标文件具有相同的格式，但是它不包含未解决的引用。

7.2.4 代码加载

现在可执行文件已经存在于磁盘中，操作系统就可以将其读入内存并启动执行该程序，在UNIX系统中，**加载器(Loader)**按照一下步骤工作：

1. 读取可执行文件头来确定代码段和数据段的大小
2. 为正文和数据创建一个足够大的地址空间
3. 将可执行文件中的指令和数据复制到内存中
4. 把主程序的参数复制到栈顶
5. 初始化机器寄存器，将栈指针指向第一个空位置
6. 跳转到启动例程，它将参数复制到参数寄存器并且调用程序的main函数，当main函数返回时，启动例程通过系统调用exit终止程序

接下来，就让我们一起构造我们自己的汇编器和链接器！

7.3 汇编器

7.3.1 汇编器原理详解

正如上文所述，汇编器将汇编语言文件翻译成二进制机器指令和二进制数据组成的文件，我们将在下文阐述MIPS汇编器的实现细节。
首先我们先来看看汇编器在处理过程中可能面临的各种情况：

1. 简单情况：
    - 计算和逻辑指令，例如`addu $rd, $rs, $rt`、`mult $rs, $rt`、`lw $rt, offset($rs)`等
    - 所有的需要的，将其汇编成二进制指令的信息都已经包含在了指令中
2. 分支指令(Branch)：
    - 分支指令中的imm字段代表的是相对当前指令地址的偏移量，即需要的是相对地址
    - 一旦将所有的伪指令解析成实际的机器代码以后，由于MIPS是定长指令，因此通过简单的计算我们便可以得到实际的地址偏移量
3. 前向引用：汇编语言允许标签在定义之前就被使用，如下图所示：

```
    beq $zero, $zero, Label
Label:
```

汇编器在看到指令beq，它不知道标签Label在哪里，因此也就无从得知相对地址偏移量了。

为解决这个问题，我们首先引入一个概念 **“遍(Pass)”**。它指的是对源程序（包括源程序的中间形式）从头到尾扫描一次，并做有关的加工处理，生成新的源程序中间形式或目标程序，通常称为遍。
汇编器通常采用两遍的方式来处理可能出现的前向引用：

- 第一遍，汇编器将汇编文件的每一行读入，如果一行以标签作为开始，汇编器在他的符号表中记录标签的名字以及在文件中的位置
- 第二遍，汇编器使用整个文件的符号表中的信息，在这一遍产生机器代码。若依然有未决的引用，说明该引用是外部文件定义的，将由链接器来进行解析

4. 跳转指令(Jump)：例如j, jal等指令
    - 这类跳转指令需要跳转的绝对地址，汇编器对于单一文件的操作，不可能得到最终的绝对地址
5. 数据引用：例如la等指令：
    - la指令将由汇编器处理为lui和ori指令，同样需要32位的绝对地址

可以看到，我们对于需要绝对地址的指令以及外部引用都是无法解析的，因此汇编器将创建两张表。

1. **符号表(Symbol Table)**：每个文件都有一个符号表，用来记录可能被其他文件引用的：
    - 标签(Label)：跳转标签，函数定义
    - 数据：出现在文件`.data`段的数据标签
2. **重定位表(Relocation Table)**：需要在之后的过程中需要得到地址的标签（当前无法确定的）：
    - j, jal等指令的跳转标签，无论是文件内定义的还是文件以外定义的
    - 任何数据标签，例如任何被la指令引用的数据标签

7.3.2 汇编器具体实现

汇编器实现概述

本次实验，我们的汇编器将实现一个MIPS指令的子集。同时我们的汇编器是一个如上文描述的“两遍”汇编器，实现汇编.data和.text段。它将按照以下流程工作：
第一遍：读取输入的.s文件

举例如下：

```
.data
num: .space 8

.text
add:
    lw $s5, -4($sp)
    lw $s6, -8($sp)
    addu $t0, $s5, $s6
    move $v0, $t0
    jr $ra
main:
    la $v1, num
    li $s1, 10
    li $s2, 20
    sw $ra, 0($a3)
    sw $s1, -4($a3)
    sw $s2, -8($a3)
    move $sp, $a3
    jal add
    lw $ra, 0($sp)
    move $t0, $v0
    beq $t0, $0, end
    j end
end:
```

- 剔除文件注释（在.s文件中，注释为#开头的单行注释）

- 扩展伪指令
- 记录文件符号，构建相应的符号表
  - 在我们的处理中，为了区分符号是来时.data还是来自.text中，我们为来自.data段中的符号，在最开始追加一个`%`用于在符号表中区分。
- 标签（是否重复定义）和伪指令（是否合法）将被检查确认
- 输出：中间文件（.int），对于.data段，只需要输出.data段所占据的字节数即可，按照如下格式。

```
.data
8

.text
lw $s5 -4 $sp
lw $s6 -8 $sp
addu $t0 $s5 $s6
addu $v0 $at $t0
jr $ra
lui $at num@Hi
ori $v1 $at num@Lo
addiu $s1 $0 10
addiu $s2 $0 20
sw $ra 0 $a3
sw $s1 -4 $a3
sw $s2 -8 $a3
addu $sp $at $a3
jal add
lw $ra 0 $sp
addu $t0 $at $v0
beq $t0 $0 end
j end

```

第二遍：读取.int中间文件

- 将指令翻译为机器代码，对于需要重定位的字段，一律进行**清零**处理
- 指令语法和参数检查
- 构建重定位表
- 输出：目标文件（.out），其中包括数据段大小(.data)，机器代码(.text)，符号表(.symbol)，重定位表(.relocation)

```
.data
8

.text
8fb5fffc
8fb6fff8
02b64021
00281021
03e00008
3c010000
34230000
2411000a
24120014
acff0000

```

```
acf1fffc
acf2fff8
0027e821
0c000000
8fbf0000
00224021
11000001
08000000

.symbol
0   %num
0   add
20  main
72  end

.relocation
20  num@Hi
24  num@Lo
52  add
68  end

```

汇编指令集

汇编器**支持**如下的寄存器符号：
`$0 - $31, $zero, $at, $v0 - $v1, $a0 - $a3, $t0 - $t9, $s0 - $s7, $k0 - $k1, $sp, $fp, $ra`。

汇编器需要支持22条MIPS基本指令，和5条伪指令，如下（关于指令的详细介绍，请参考MIPS指令文档或MARS帮助文档）：

指令	格式	指令	格式
Add Usigned	`addu $rd, $rs, $rt`	Or	`or $rd, $rs, $rt`
Set Less Than	`slt $rd, $rs, $rt`	Set Less Than Unsigned	`sltu $rd, $rs, $rt`
Jump Register	`jr $rs`	Shift Left Logical	`sll $rd, $rt, shamt`
Add Immediate Unsigned	`addiu $rt, $rs, imm`	Or Immediate	`ori $rt, $rs, imm`
Load Upper Immediate	`lui $rt, imm`	Load Byte	`lb $rt, offset($rs)`
Load Byte Unsigned	`lbu $rt, offset($rs)`	Load Word	`lw $rt, offset($rs)`
Store Byte	`sb $rt, offset($rs)`	Store Word	`sw $rt, offset($rs)`
Branch on Equal	`beq $rs, $tr, label`	Branch on Not Equal	`bne $rs, $rt, label`
Jump	`j label`	Jump and Link	`jal label`
Mult	`mult $rs, $rt`	Div	`div $rs, $rt`
Move from $hi	`mfhi $rd`	Move from $lo	`mflo $rd`

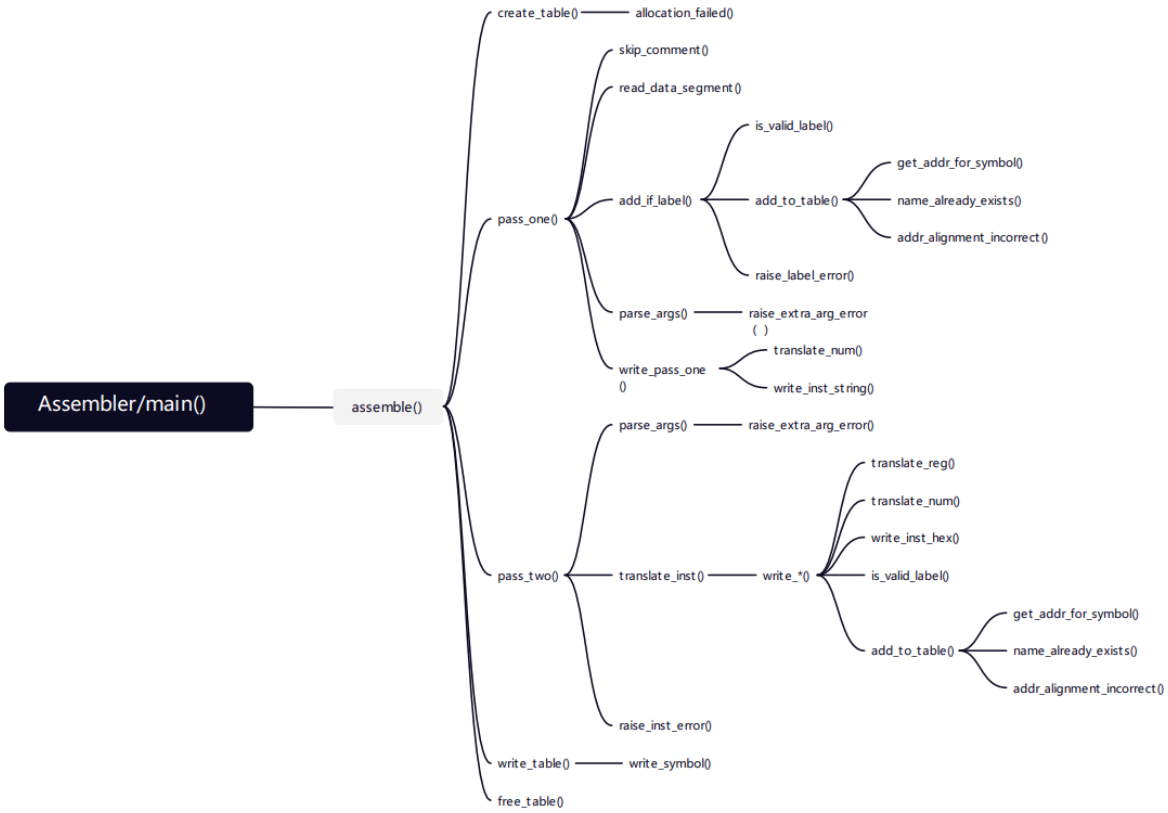
伪指令	格式
Load Immediate	`li $rt, imm`
Move	`move $rd, $rs`
Divides and return reminder	`rem $rd, $rs, $rt`
Branch if greater than or equal	`bge $rs, $rt, label`
Branch if not equal to zero	`bnez $rs, label`
Load address	`la $rt, label`

汇编器支持.data段如下声明：（注释中也有解释）

```
Label: .space number_of_bytes
```

汇编器实现

汇编器整体调用关系如下：



为了简化实验步骤，我们提供了部分汇编器源文件，大家只需要按照以下步骤实现相关函数，但是为了便于理解，还请大家通读一遍程序(相关程序为assembler.c, /lib/*.h, /assembler-src/*.h)，各个源文件都提供了详尽的注释说明。

所有需要你完成的函数都已经声明在my_assembler_utils.h文件中。对应的函数请在my_assembler_utils.c中实现，最终**仅提交my_assembler_utils.c**文件即可。

你可以在文件根目录下，在mingw或linux等环境中，通过`make assembler`命令行来编译自己的汇编器，将会在根目录下得到assembler可执行文件。

在上文中我们说明了汇编器是如何构建符号表和重定位表的，在我们的汇编器中也定义了与之相对应的数据结构，它是我们实现构建查找符号表的基础：

```
typedef struct {
    Symbol *tbl;    //符号数组
    uint32_t len;   //符号表长度，即tbl中当前存在的符号数目
    uint32_t cap;   //符号表当前最大容量
    int mode;       //符号表模式
} SymbolTable;      //相当于手动实现的Vector
```

每个符号用一个`Symbol`结构体表示：

```
typedef struct {
    char *name;     //符号名称
    uint32_t addr;  //符号在该文件中的相对偏移量（以字节为单位）
} Symbol;
```

符号表共有两种模式，也定义在table.h中：

- `SYMTBL_NON_UNIQUE`：允许名称重复出现在符号表中（符号在跳转，分支指令中可以出现多个）

- `SYMTBL_UNIQUE_NAME`：不允许名称重复出现在符号表中（符号定义只能有一个）

Step 1
实现函数

```
int read_data_segment(FILE *input, SymbolTable *symtbl);
```

该函数用于汇编器第一遍中，读取.data段声明的各个符号，将其保存在符号表中，最后返回.data段所占据的字节数

step2

实现函数

```
int add_to_table(SymbolTable *table, const char *name, uint32_t addr);
```

该函数将一个符号名及其偏移量保存在传入的符号表中。需要注意的是，判断该函数是否重复。

step 3
实现部分机器指令到机器码的转换，函数：

```
int write_lui(uint8_t opcode, FILE *output, char **args, size_t num_args,
uint32_t addr, SymbolTable *reltbl);
```

该函数将第一遍获得的lui指令翻译为机器码，注意有以下两种情况

```
lui $reg, 100
lui $reg, Label@Hi/Lo
```

也就是说，lui可能是加载地址高16位，也可能是将立即数加载到寄存器的高位。

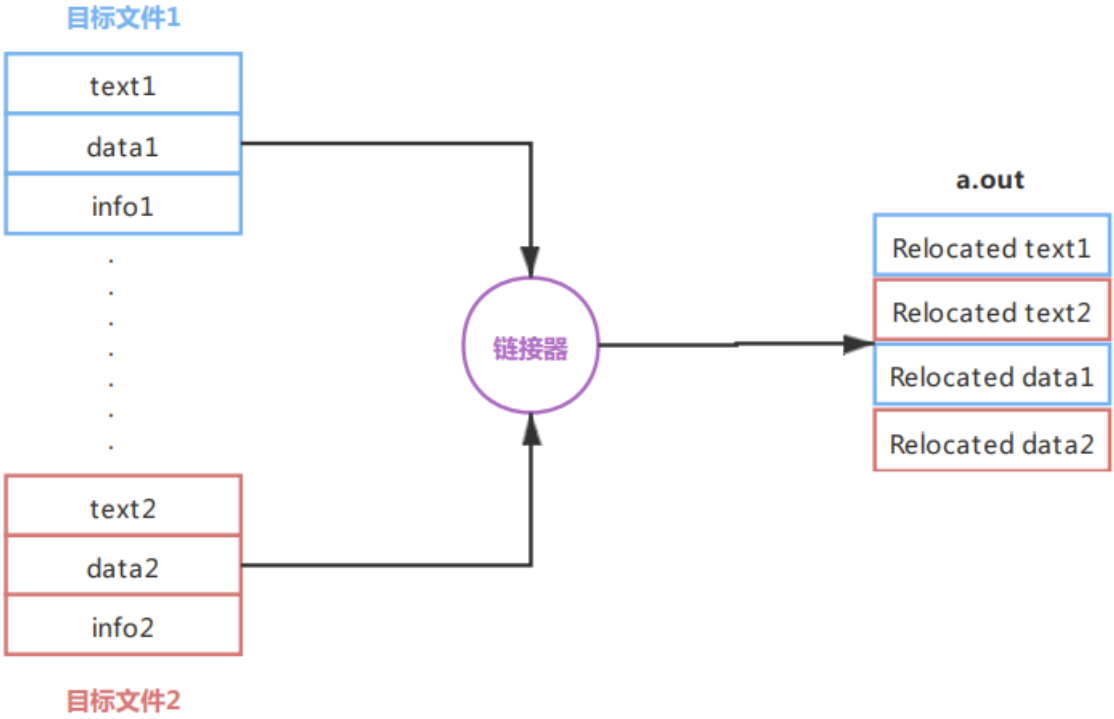
代码提交

在完成my_assmebler_utils.h文件中的所有函数以后以后，就可以提交**my_assembler_utils.c**

7.4 链接器

7.4.1 链接器原理详解

链接器读取我们汇编器输出的数个.o文件，利用目标文件中的信息，进行符号解析，最终输出可执行的机器代码，例如（a.out）。



对于我们的链接器而言，有四类地址：

1. PC相对地址(beq, bne)：不需要重定位（位置无关的）
2. 绝对地址(j, jal)：需要重定位
3. 外部引用(jal)：需要重定位
4. 数据引用(lui, ori)：也就是la, 需要重定位

链接器能够得到每个文件的text和data段的大小以及它们的顺序。利用这些，链接器便可以计算每个符号的绝对地址。
为了解析符号，链接器首先搜索每个文件的符号表（即每个文件定义的符号），如果没有找到，将继续在库文件中搜索（例如printf）。一旦获取到符号的定义位置，链接器就会填入实际的地址，完善机器代码。最终链接器输出的可执行文件包括.text和.data段（加上ELF文件头）。

7.4.2 链接器具体实现

汇编器实现概述

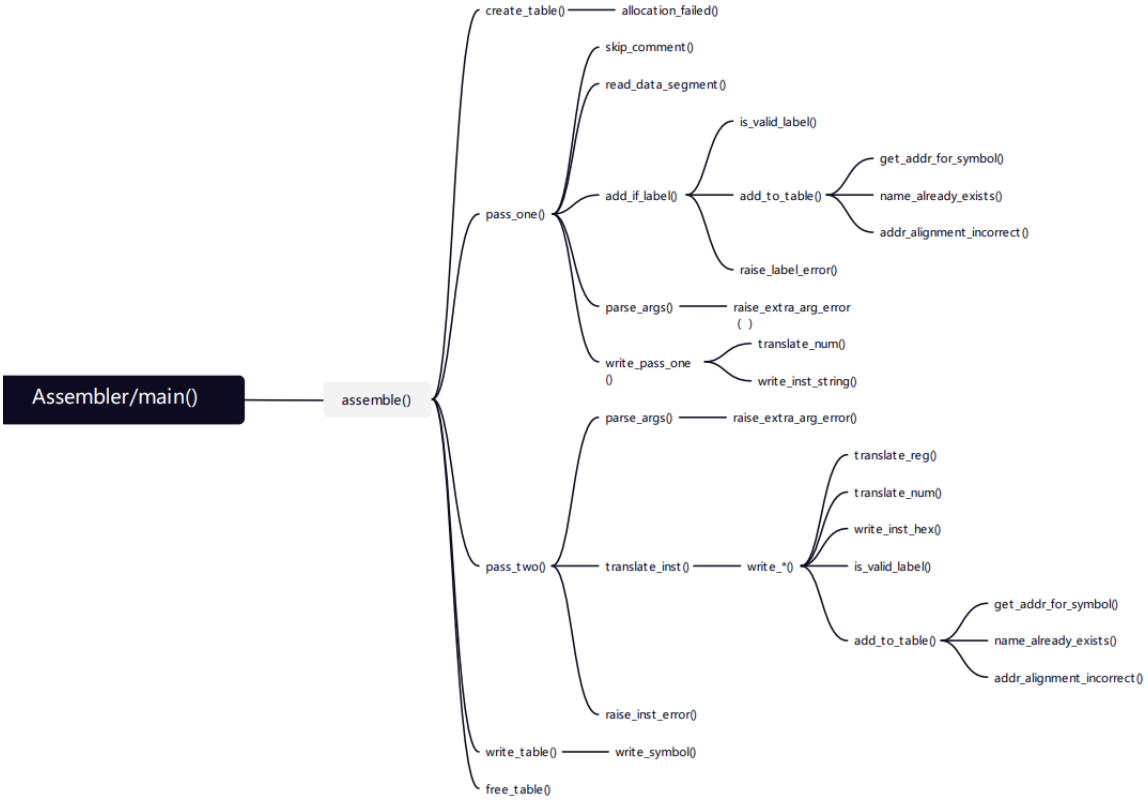
我们在汇编器中，解析了.data，.text段，因此我们在这里也将实现一个简化过的链接器，他将把一个或多个.o文件的.data和.text合并在一起来创建可执行的机器代码，但是我们不会为其添加ELF文件头，因此你可以认为我们的输出是可执行文件去掉ELF文件头以后的二进制bin文件。
具体的链接步骤如下：

1. 创建一个空的全局符号表，他将存储每个符号（定义位置）的绝对地址，因此每个符号至多出现一次。
2. 对于每个目标文件都创建一个独立的重定位表，将包含每个需要重定位符号的相对地址
3. 打开每个目标文件，分别读取它的.data，.text, .symbol, .relocation段：
    - 如果是.data段，读取该文件.data段的大小，用于计算上述提到的绝对地址
    - 如果是.text段，计算指令数目，得到该文件的指令将会占用多少字节
    - 如果是.symbol段，将其读入，将相对地址转换为绝对地址（怎么做？），并合并进第1步创建的全局符号表
    - 如果是.relocation段，将其读入进该文件的重定位表
4. 打开输出文件

5. 再次打开每个目标文件，读取.text段，检查每条指令是否需要重定位，如果需要，使用重定位表和符号表查找符号的绝对地址，并填入指令的相应字段。最终输出即可。

链接器实现

链接器整体调用关系如下：



我们在汇编器的重定位表中引入新的结构体RelocData（linker_utils.h）：
主要是在重定位表中加入字段text_size用于记录该文件的.text需要占据的字节数。

```
typedef struct {
    SymbolTable *table; //重定位表
    int text_size;      //相应文件.text节大小
} RelocData;
```

step1

实现函数

```
int fill_data(FILE *input, SymbolTable *symtbl, RelocData *reldt, uint32_t base_text_offset, uint32_t base_data_offset);
```

该函数计算依次读取输入文件的.data, .text, .symbol, .relocation段。

对于.data段，只需读取汇编器计算出的字节数保存在重定位表中即可。

对于.text段，需要统计指令数量，计算.text段占据的字节数

step 2

实现函数

```
int inst_needs_relocation(SymbolTable *reltbl, uint32_t offset);
```

判断当前位置（offset）的指令是否需要重定位。

在这之后，你的链接器就可以运行了，在根目录下执行指令`make linker`来编译你的链接器，将在根目录下得到`linker`可执行文件。

样例

为了便于理解，提供以下样例以供参考：

有以下两个.s文件需要进行处理：

main.s:

```
.data
num: .space 8

.text
main:
    la $v1, num
    li $s1, 10
    li $s2, 20
    sw $ra, 0($a3)
    sw $s1, -4($a3)
    sw $s2, -8($a3)
    move $sp, $a3
    jal add
    lw $ra, 0($sp)
    move $t0, $v0
    beq $t0, $0, end
    j end
end:
```

add.s:

```
.data

.text
add:
    lw $s5, -4($sp)
    lw $s6, -8($sp)
    addu $t0, $s5, $s6
    move $v0, $t0
    jr $ra
```

则在汇编结束后分别得到：

main.out:

```
.data
8

```

```
.text
3c010000
34230000
2411000a
24120014
acff0000
acf1fffc
acf2fff8
0027e821
0c000000
8fbf0000
00224021
11000001
08000000

.symbol
0   %num
0   main
52  end

.relocation
0   num@Hi
4   num@Lo
32  add
48  end

```

add.out:

```
.data
0

.text
8fb5fffc
8fb6fff8
02b64021
00281021
03e00008

.symbol
0   add

.relocation

```

链接器指令命令：`linker main.out add.out main.o`，最终链接以后的文件为main.o，如下：

```
3c011001
34230000
2411000a
24120014
acff0000
acf1fffc
acf2fff8
0027e821
0c10000d

```

```
8fbf0000
00224021
11000001
0810000d
8fb5fffc
8fb6fff8
02b64021
00281021
03e00008
```

代码提交

在完成my_linker_utils.h文件中的所有函数以后，即可提交**my_linker_utils.c**文件进行评测。