Project 1

一、程序说明：

1.In general, due to the limitation of C1 programming language, it is tricky to implement some algorithms such as quick sort, so I choose a much easier algorithm called selection sort.
2.As it only support “while” loop and I am used to use “for” loop in my program, it is easy to forget to increase the iterators, which leads to crucial bugs in the program(infinite loop). As I get used to it, it is more suitable to write a program only with while loop.
3.To solve the limitation that the functions do not have any parameter, I write the program with only main function for gcd.c and prime.c, while for sort.c, I declare the array as a global variable, so I do not to pass it to the sort function as a parameter.

prime.c :

This program is designed to calculate the prime numbers smaller than 30, using the algorithm called sieve of Eratosthenes.
It is designed to test the global const variables, while loop and the definition of array. I mostly use “!=” instead of other relation operators to test it specifically. And in the last while loop, I test the if condition without use any relation operator.

sort.c:

In this program, I implement selection sort to sort an array with descending order. It is a very easy algorithm, just two while loops and easy to test.
It is mainly use to test the function call, as you can see from the source code, I separate the main sort module from the main function and create a new function called sort.
In the meantime, I also use expressions and negative numbers in the definition of the array, so it can also test some basic operators such as '+', '-', '*', and '/' .

gcd.c:

It is also a very easy program, used to calculate the greatest common divisor of two integers. It uses the Euclidean algorithm, just a small while loop.
As it has to use the % operator, it is a good example to test it.
It is also designed to test the “if …. else … “.

note: for this project, I use gcc as my compiler so that I do not have any constant variable declaration using any declared constant variables.

二、相关问题的回答

预处理：

命令： cpp *.c 产生 *.i 文件
gcc 首先调用 cpp 对源程序进行处理，主要在三个方面：

1. 将包含的头文件中的内容展开到源文件中
2. 进行宏展开，即将宏定义的内容填充到使用的地方
3. 处理条件编译

总之，预处理只是对源文件简单的处理，特别是将包含的头文件展开，减少了后续的工作量

汇编器：

命令： gcc -S *.c 产生 .s 结尾的汇编代码
其主要工作有：

创建符号表， 对所有变量分配地址并将源文件翻译成汇编代码，最简单的汇编器对输入扫描两遍，一遍形成符号表并分配空间，第二遍将源代码翻译成机器码

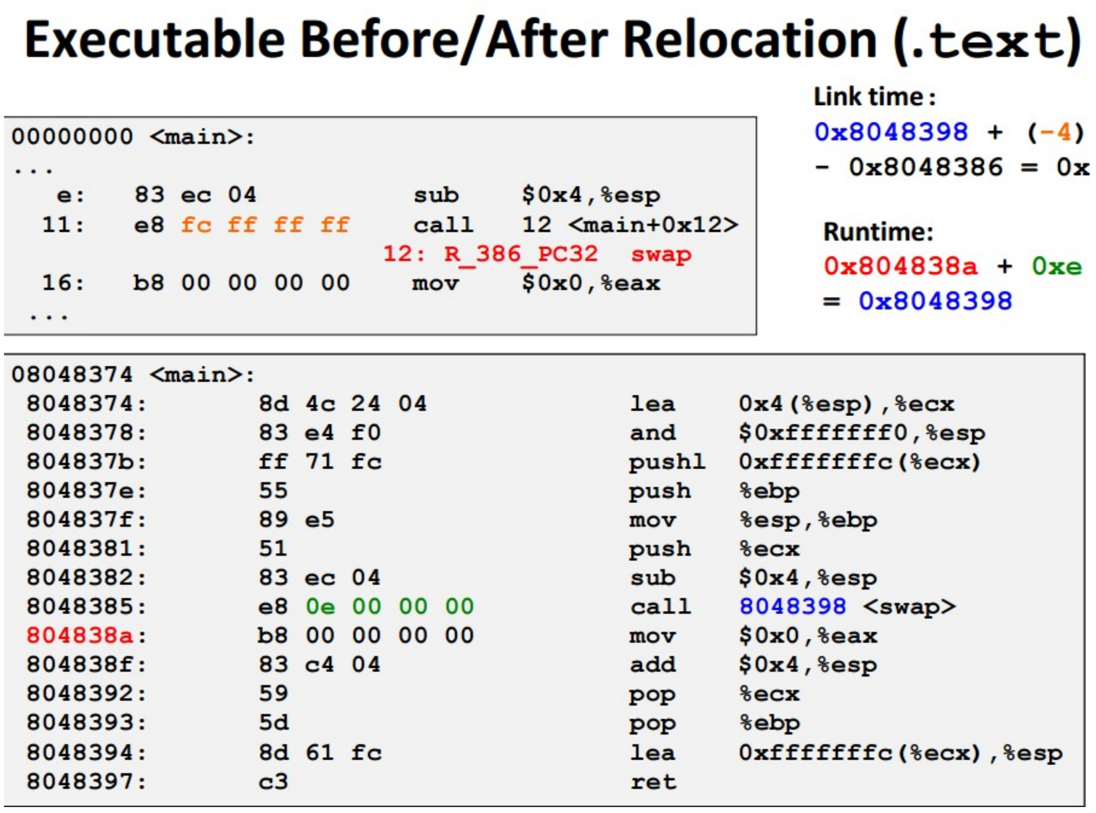
连接器：

命令： ld objfile.... 产生可执行的目标文件

连接分为动态和静态两种，其中静态连接器将多个可重定位的文件组合成一个可执行的目标文件，而动态连接器则可以在运行时进行连接。 其主要进行符号解析和重定位工作.

符号解析：

编译时编译器向汇编器输出的全局符号分为强弱两种，函数和已初始化的全局变量为强符号，未初始化的全局变量为弱符号。unix 连接器不允许多重强符号定义，强弱符号同时定义时选择强符号，多个如符号定义时选择任意一个。 为了防止出现这样的连接问题， 尽量少用全局变量。



Executable Before/After Relocation (.text)

```
00000000 <main>:
...
   e:   83 ec 04          sub    $0x4,%esp
  11:   e8 fc ff ff ff    call   12 <main+0x12>
                     12: R_386_PC32  swap
  16:   b8 00 00 00 00    mov    $0x0,%eax
 ...
```

Link time:

```
0x8048398 + (-4)
- 0x8048386 = 0x
```

Runtime:

```
0x804838a + 0xe
= 0x8048398
```

```
08048374 <main>:
 8048374:       8d 4c 24 04         lea    0x4(%esp),%ecx
 8048378:       83 e4 f0            and    $0xfffffff0,%esp
 804837b:       ff 71 fc            pushl  0xfffffffc(%ecx)
 804837e:       55                  push   %ebp
 804837f:       89 e5               mov    %esp,%ebp
 8048381:       51                  push   %ecx
 8048382:       83 ec 04            sub    $0x4,%esp
 8048385:       e8 0e 00 00 00      call   8048398 <swap>
 804838a:       b8 00 00 00 00      mov    $0x0,%eax
 804838f:       83 c4 04            add    $0x4,%esp
 8048392:       59                  pop    %ecx
 8048393:       5d                  pop    %ebp
 8048394:       8d 61 fc            lea    0xfffffffc(%ecx),%esp
 8048397:       c3                  ret
```

(source: 15-213@CMU)

这张图可以清晰的显示出重定位前后的汇编代码的状态。其调用的是 swap 函数与 main 函数不在同一个文件中。在重定位前，main 函数预留了一个指令，一开始是调用自己的，直到重定位时，其算出需要调用函数的相对地址，然后修改其中的地址就可以了，这样的处理方式大大增加了连接的效率。

我们往往需要调用一些常用的函数，而连接器需要将其与我们所写的程序连接起来，一开始使用的静态库(.a 文件)，连接器仅将需要的函数与我们的程序连接。而这样会造成一些问题，比如每个程序都要连接一个造成内存和外存的空间浪费，库的更新成本过高等。当今主要使用的是共享库(.so 文件)，其可以在加载时或者运行时连接，可以被多个程序共享，提高了内存利用率。

llvm 命令：

llc :

llvm static compiler

其将源文件编译为特定架构的汇编代码

llvm-link:

其将多个 LLVM bitcode 文件连接成一个 LLVM bitcode 文件

llvm-as:

llvm 的汇编器，其将汇编码转化为 LLVM bitcode.

lli:

其运行 LLVM bitcode 格式的文件.

三、实验小结

本次实验主要是准备工作，比较简单，不过也学到了很多新的东西，比如 git 的基本使用，makefile 的编写等(中途转系生比较渣_(:з」∠)_)。

PB13203131
陈安哲