Lab2 系统编程

一、 实验概述

本次实验中，我们会了解基本的系统编程知识，包括C语言介绍，内联汇编，C语言编译和链接，Makefile。

二、实验目的

1. 了解C语言的基本语法和编译链接过程
2. 了解C语言中内联汇编的使用
3. 使用Makefile进行自动化编译

三、实验内容及相关知识点

第一部分. C语言

从hello world开始

首先我们在桌面右键打开Terminal，然后创建一个文件夹，进入文件夹，我们创建一个C语言文件

```
mkdir OSlab_C_Programming
cd OSlab_C_Programming/
vim hello.c
```

接着我们在vim编辑器中输入第一段C语言代码

```
#include <stdio.h>

int main(){
    printf("Hello, World!\n");
    return 0;
}
```

现在我们没有IDE，我们要怎么运行我们的C语言程序呢?

运行hello.c

当我们编译一个C程序时，系统进行预处理，编译和优化，汇编，链接，最终生成一个可执行文件，我们才能运行。

这里我们要使用gcc工具，我们可以使用man命令查看gcc命令的参数和介绍

> `gcc` (GNU compiler Collection) is a open-source compiler system
>
> DESCRIPTION
>
> When you invoke GCC, it normally does preprocessing, compilation, assembly and linking. The "overall options" allow you to stop this process at an intermediate stage. For example, the -c option says not to run the linker. Then the output consists of object files output by the assembler.

> SYNOPSIS
>
> **-c** Compile or assemble the source files, but do not link.
>
> **-S** Stop after the stage of compilation proper; do not assemble.
>
> **-E** Stop after the preprocessing stage; do not run the compiler proper.
>
> **-o filename** Place output in file file.
>
> If no parameters, gcc will do all things and output an execute file a.out

```
# 方法1 直接用无参数的形式
gcc hello.c  #会生成a.out
# 使用file命令查看a.out的文件类型
file a.out
# 运行a.out
./a.out
```

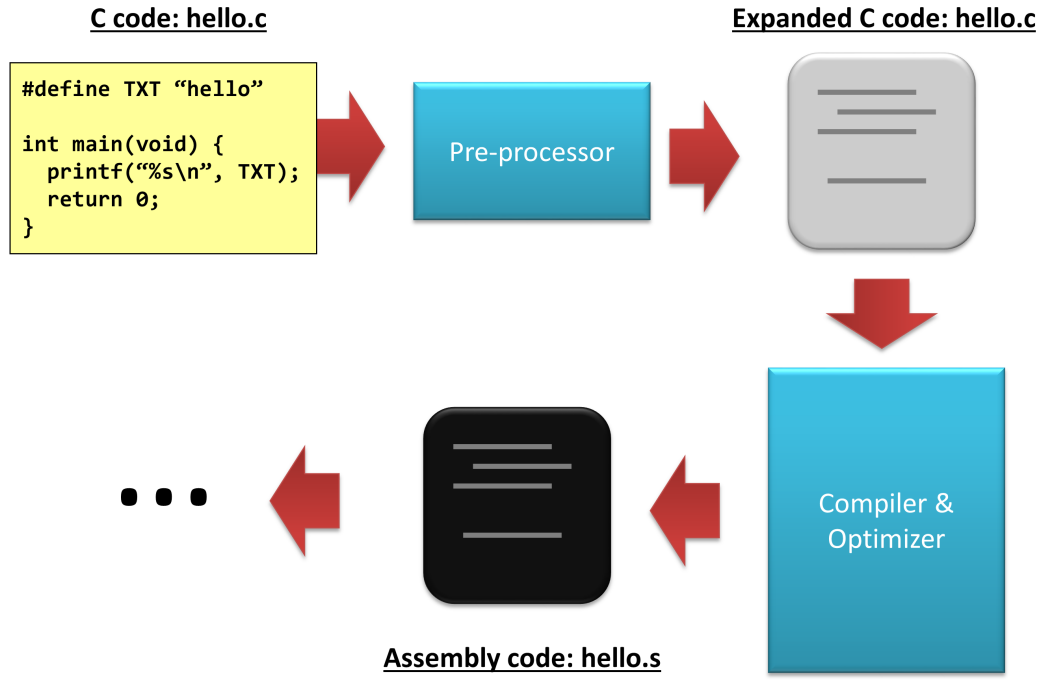
```
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ gcc hello.c
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ file a.out
a.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linke
d, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=8cba3c632621b559ed192f
211dd03b5056bf8d9a, for GNU/Linux 3.2.0, not stripped
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ ./a.out
Hello, World!
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$
```

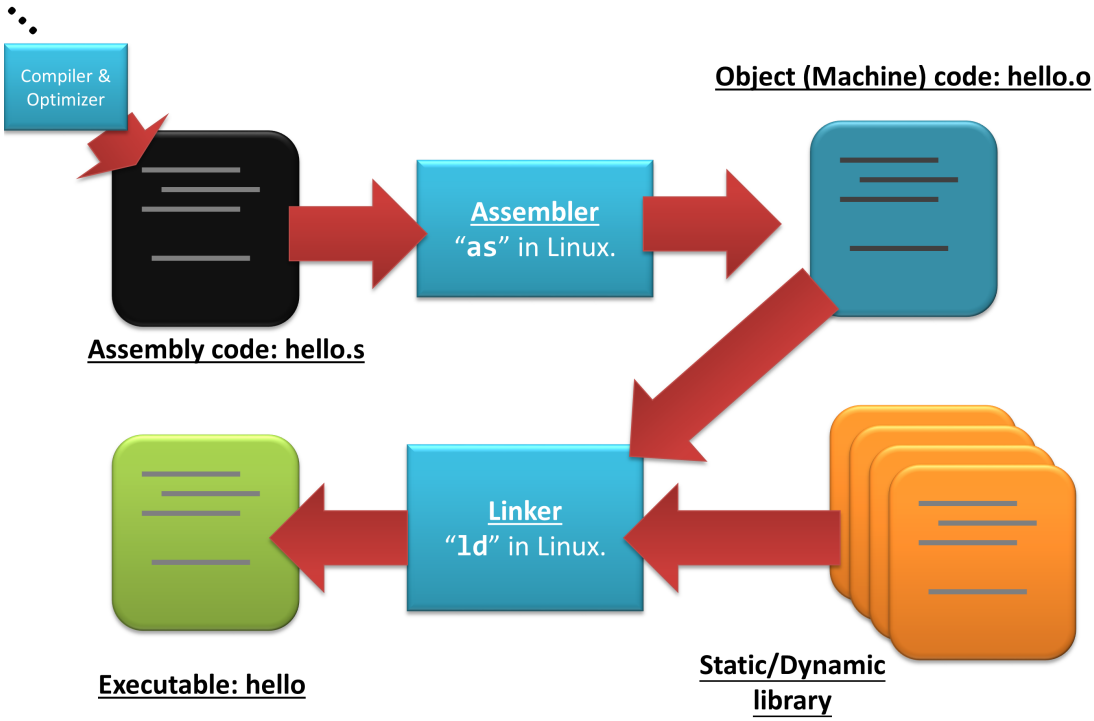
```
# 方法2 使用-o参数
gcc -o hello hello.c  #会生成名为hello的可执行文件
# 使用file命令查看hello的文件属性
file hello
# 运行hello
./hello
```

```
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ rm a.out -rf
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ ls
hello.c
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ gcc -o hello hello.c
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ file hello
hello: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linke
d, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=8cba3c632621b559ed192f
211dd03b5056bf8d9a, for GNU/Linux 3.2.0, not stripped
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ ./hello
Hello, World!
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$
```

从.c 文件到可执行文件的过程

(本部分参考：(https://calvinkam.github.io/csci3150-Fall17-lab3/building-a-program.htm)





预处理

在这一部分，#define，#include等一些宏被拓展为C代码。

编译和优化

编译器的工作是将可读的代码转换成机器可以理解的代码。

1. 检查代码的语法并且进行分析
2. 如果1中没有错误，将产生中间代码，比如：汇编代码

> 机器代码：直接在机器上执行的代码，人类不可读
>
> 汇编代码：人类可阅读形式的机器代码

优化器：顾名思义，对代码进行优化。

汇编和链接

> 汇编器将汇编代码转换为目标代码（机器代码）。
>
> 链接器将所有目标文件和库文件结合在一起生成可执行文件。

第二部分. Make 和 Makefile介绍

考虑一下，如果我们的工程稍微大一点（比如包含多个C语言文件），每次运行一次我们都要执行很多次gcc命令，是否有一种编译工具可以简化这个过程呢？接下来我们介绍自动化编译工具make。

考虑我们有三个文件

```
//print.h 头文件
#include <stdio.h>
void print(void);

//print.c
#include "print.h"
void print(){
    printf("Hello, World!\n");
}

//main.c
#include "print.h"
int main(){
    print();
    return 0;
}
```

因为文件中的依赖关系，我们需要为每个.c文件生成.o目标文件，然后把两个.o文件生成可执行文件：

```
gcc -c main.c
gcc -c print.c
gcc -o main main.o print.o

./main
```

```
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ ls
main.c  Makefile  print.c  print.h
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ gcc -c main.c
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ gcc -c print.c
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ ls
main.c  main.o  Makefile  print.c  print.h  print.o
```

```
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ gcc -o main main.o pr
int.o
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ ls
main  main.c  main.o  Makefile  print.c  print.h  print.o
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ ./main
Hello, World!
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$
```

如果我们的文件数量很多，每次运行就会变得十分的复杂。为了使整个编译过程更加容易，可以使用Makefile。

Makefile

```
main : main.o print.o
    gcc -o main main.o print.o
main.o : main.c print.h
    gcc -c main.c
print.o : print.c print.h
    gcc -c print.c
clean:
    rm main main.o print.o
```

我们在整个文件夹新建一个Makefile文件，把上面的内容放进去，这样我们只需要执行一句make命令，就可以完成整个编译过程。

```
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ ls
main.c  Makefile  print.c  print.h
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ make
gcc -c main.c
gcc -c print.c
gcc -o main main.o print.o
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ ls
main  main.c  main.o  Makefile  print.c  print.h  print.o
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$ ./main
Hello, World!
oslab@oslab-virtual-machine:~/Desktop/OSlab_C_Programming$
```

Makefile的基本结构

```
target: dependencies
[tab] system command
```

Makefile工作原理

参考：（makefile介绍 — 跟我一起写Makefile 1.0 文档 (seisman.github.io)）

在默认的方式下，也就是我们只输入 `make` 命令。那么，

1. make会在当前目录下找名字叫“Makefile”或“makefile”的文件。
2. 如果找到，它会找文件中的第一个目标文件（target），在上面的例子中，他会找到“main”这个文件，并把这个文件作为最终的目标文件。
3. 如果main文件不存在，或是main所依赖的后面的 `.o` 文件的文件修改时间要比 `main` 这个文件新，那么，他就会执行后面所定义的命令来生成 `main` 这个文件。
4. 如果 `main` 所依赖的 `.o` 文件也不存在，那么make会在当前文件中找目标为 `.o` 文件的依赖性，如果找到则再根据那一个规则生成 `.o` 文件。（这有点像一个堆栈的过程）
5. 当然，你的C文件和H文件是存在的啦，于是make会生成 `.o` 文件，然后再用 `.o` 文件生成make的终极任务，也就是执行文件 `main` 了。

make clean

通过上述分析，我们知道，像clean这种，没有被第一个目标文件直接或间接关联，那么它后面所定义的命令将不会被自动执行，不过，我们可以显式要make执行。即命令——`make clean`，以此来清除所有的目标文件，以便重新编译。

更多关于Makefile的知识请查看：（跟我一起写Makefile 1.0 文档）

第三部分. 内联汇编介绍

内联函数：在C语言中，我们可以指定编译器将一个函数代码直接**复制**到调用其代码的地方执行。这种函数调用方式和默认压栈调用方式不同，我们称这种函数为内联函数。内联函数降低了函数的调用开销。在 C语言中，可以使用`inline`关键字将函数标记为内联函数。

```
// 一个C语言内联函数的声明与实现
int sum(int a, int b);  //声明，在ubuntu使用的gcc如不加此行声明需增加-O编译选项
inline int sum(int a, int b)   //实现，在实现前加inline关键字则为内联函数
{
   return a + b;
}

//调用上述内联函数
int main(){
    printf("%d\n", sum(2, 3));
    return 0;
}
```

内联汇编：内联汇编相当于用汇编语句写成的内联函数。它方便，快速，对系统编程有着举足轻重的作用。

riscv下的gcc内联汇编语法规则：

```
asm volatile(
    汇编指令列表
    ：输出操作数 //非必需
    ：输入操作数 //非必需
    ：可能影响的寄存器或存储器 //非必需
);

__asm__  __volatile__(
    "Instruction_l;\
     Instruction 2;\
     ...\
     Instruction_n;"
    :[out1]"=r"(value1), [out2]"=r"(value2), ...[outn]"=r"(valuen)
    :[in1]"r"(value1),[in2]"r"(value2),...[inn]"r"(valuen)
    :"r0","r1",..."rn"
);
```

Explanation:

1. `asm` 是GCC的关键字，表示内联汇编操作，注 `__asm__` 是GCC中`asm`的宏定义，也可以使用
2. `volatile` 或者`__volatile__`是可选的，表示不进行任何优化，否则某些操作可能会被优化掉。
3. 汇编指令列表， 即要嵌入的汇编指令。每条指令应该被双引号括起来作为字符串，两条指令之前必须以 \n 或者 ; 作为分隔符，没有添加分隔符的两个字符串将会被合并成为一个字符串。
4. 输出操作数，用来指定当前内联汇编程序的输出操作符列表
5. 输入操作数，用来指定当前内联汇编语句的输入操作符列表
6. 可能影响的寄存器或存储器，用于告知编译器当前内联汇编语句可能会对某些寄存器或内存进行修改，使得编译器在优化时将其因素考虑进去

```
// 在riscv中，ebreak指令会触发断点中断
// 可以使用内联汇编在C语言文件中触发断点中断
asm volatile("ebreak");
```

```
//这是我们下节课中会用到的一段内联汇编，下节课会进行详细介绍
// 这段代码的作用主要是分别在x17,x10,x11,x12寄存器设置对应的参数，调用ecall指令，完成一定的功能，比如打印一个字符，最后接收返回值。寄存器作用可查看RISCV汇编手册：
https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md
uint64_t sbi_call(uint64_t sbi_type, uint64_t arg0, uint64_t arg1, uint64_t
arg2) {
    uint64_t ret_val;  // 定义一个返回值
    __asm__ volatile (
        "mv x17, %[sbi_type]\n"
        "mv x10, %[arg0]\n"
        "mv x11, %[arg1]\n"
        "mv x12, %[arg2]\n"
        "ecall\n"
        "mv %[ret_val], x10"
        : [ret_val] "=r" (ret_val)
        : [sbi_type] "r" (sbi_type), [arg0] "r" (arg0), [arg1] "r" (arg1),
[arg2] "r" (arg2)
        : "memory"
    );
    return ret_val;
}
```

此部分参考（(riscv下的GCC内联汇编)

RISCV汇编手册：https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md

更多内联汇编知识：

- GCC Manual， 版本为5.0.0 pre-release,6.43节（How to Use Inline Assembly Language in C Code）
- GCC-Inline-Assembly-HOWTO

四、下一实验简单介绍

下一次实验，我们将正式进入我们的操作系统内核，一个简单的最小化内核。