# Makefile 讲解

by 郑玉典

关于 Makefile 的基本介绍大家可以从网上简单的了解到，Windows 下与 Linux 下下载专门的版本都能进行使用，Windows 下需要安装 MINGW（Windows 下的 gcc 编译工具）或者更强大的 cygwin(Windows 平台上模拟 UNIX 运行环境，可以选择相应组件进行安装)，但是 Windows 下需要自己将安装的地点添加到环境变量中，以便在 cmd 中可以直接进行 make 使用。

Makefile 的核心是“自动化编译”，简简单单的 make 命令就可以使很庞大的系统按照我们自定义的依赖规则和执行命令的顺序进行编译。

*Makefile 最重要的语法：

```
target:prerequisites
        command
```

其中有两条规则：

1. 要得到 target,需要执行命令 command
2. target 依赖 prerequisites,当 prerequisites 中至少有一个文件比 target 文件要新的时候，command 才被执行

例如：

```
boot/boot.bin :
boot/boot.asm  boot/include/load.inc boot/include/fat12hdr.inc
$(ASM) $(ASMBFLAGS) -o $@ $<
```

最后一行对应的指令：

```
nasm -I boot/include -o  boot/boot.bin
boot/boot.asm  boot/include/load.inc boot/include/fat12hdr.inc
```

如果没有进行特殊的设置,make 命令默认执行当前文件夹下 Makefile 文件中部最先定义的语句，例如在我们的 Makefile 文件下敲击 make 默认执行 nop，即

```
@echo "why not \`make image' huh? :)"
```

而我们如果想要进行编译等操作可以敲击 make image,则执行我们定义 image 下的内容，image 下的具体内容底下有相应的解释。

如果我们文件夹下存在 boot/boot.bin 而且后面 prerequisites 中的一系列文件比 boot/boot.bin 要新或者 boot/boot.bin 不存在，那么执行

```
nasm -I boot/include -o  boot/boot.bin
boot/boot.asm  boot/include/load.inc boot/include/fat12hdr.inc
```

覆盖原来的 boot/boot.bin 或直接生成 boot/boot.bin;
如果 boot/boot.bin 存在而且后面 prerequisites 中的一系列文件与 boot/boot.bin 相同那么不进行更新。

文件内容：见附件

我们的文件目录树是这样的：
（cmd 下利用 cd 命令在相应文件夹下利用 tree /F >filename.txt 可生成对应文件夹和文件的目录树）

```
|  bochsrc
|  Makefile
|
├──boot
|   |  boot.asm
|   |  loader.asm
|   |
|   └──include
|          fat12hdr.inc
|          load.inc
|          pm.inc
|
├──include
|      const.h
|      global.h
|      proc.h
|      protect.h
|      proto.h
|      sconst.inc
|      string.h
|      type.h
|
├──kernel
|      clock.c
|      global.c
|      i8259.c
|      kernel.asm
|      main.c
|      proc.c
|      protect.c
|      start.c
|      syscall.asm
|
└──lib
       klib.c
       kliba.asm
       string.asm
```

Makefile:（排版可能不是很好，建议大家看附件中的 Makefile，与这个相同）

---

```
#########################
# Makefile for Orange'S #
#########################

# Entry point of Orange'S
# It must have the same value with 'KernelEntryPointPhyAddr' in
load.inc!
ENTRYPOINT    = 0x30400

# Offset of entry point in kernel file
# It depends on ENTRYPOINT
ENTRYOFFSET   =   0x400

# Programs, flags, etc.
ASM    = nasm
DASM       = ndisasm
CC     = gcc
LD     = ld
ASMBFLAGS = -I boot/include/
ASMKFLAGS = -I include/ -f elf
CFLAGS     = -I include/ -c -fno-builtin
LDFLAGS       = -s -Ttext $(ENTRYPOINT)
DASMFLAGS = -u -o $(ENTRYPOINT) -e $(ENTRYOFFSET)

# This Program
ORANGESBOOT   = boot/boot.bin boot/loader.bin
ORANGESKERNEL= kernel.bin
OBJS       =    kernel/kernel.o    kernel/syscall.o    kernel/start.o
kernel/main.o kernel/clock.o\
          kernel/i8259.o      kernel/global.o      kernel/protect.o
kernel/proc.o\
          lib/kliba.o lib/klib.o lib/string.o
DASMOUTPUT    = kernel.bin.asm

# All Phony Targets
.PHONY : everything final image clean realclean disasm all buildimg

# Default starting position
nop :
    @echo "why not \`make image' huh? :)"

everything : $(ORANGESBOOT) $(ORANGESKERNEL)
```

```
all : realclean everything

image : realclean everything clean buildimg

clean :
    rm -f $(OBJS)

realclean :
    rm -f $(OBJS) $(ORANGESBOOT) $(ORANGESKERNEL)

disasm :
    $(DASM) $(DASMFLAGS) $(ORANGESKERNEL) > $(DASMOUTPUT)

# We assume that "a.img" exists in current folder
buildimg :
    dd if=boot/boot.bin of=a.img bs=512 count=1 conv=notrunc
    sudo mount -o loop a.img /mnt/floppy/
    sudo cp -fv boot/loader.bin /mnt/floppy/
    sudo cp -fv kernel.bin /mnt/floppy
    sudo umount /mnt/floppy

boot/boot.bin     :     boot/boot.asm     boot/include/load.inc
boot/include/fat12hdr.inc
    $(ASM) $(ASMBFLAGS) -o $@ $<

boot/loader.bin    :    boot/loader.asm    boot/include/load.inc
boot/include/fat12hdr.inc boot/include/pm.inc
    $(ASM) $(ASMBFLAGS) -o $@ $<

$(ORANGESKERNEL) : $(OBJS)
    $(LD) $(LDFLAGS) -o $(ORANGESKERNEL) $(OBJS)

kernel/kernel.o : kernel/kernel.asm include/sconst.inc
    $(ASM) $(ASMKFLAGS) -o $@ $<

kernel/syscall.o : kernel/syscall.asm include/sconst.inc
    $(ASM) $(ASMKFLAGS) -o $@ $<

kernel/start.o:  kernel/start.c  include/type.h  include/const.h
include/protect.h include/string.h include/proc.h include/proto.h \
          include/global.h
    $(CC) $(CFLAGS) -o $@ $<

kernel/main.o:   kernel/main.c   include/type.h   include/const.h
```

```
include/protect.h include/string.h include/proc.h include/proto.h \
          include/global.h
   $(CC) $(CFLAGS) -o $@ $<

kernel/clock.o: kernel/clock.c
   $(CC) $(CFLAGS) -o $@ $<

kernel/i8259.o:  kernel/i8259.c  include/type.h  include/const.h
include/protect.h include/proto.h
   $(CC) $(CFLAGS) -o $@ $<

kernel/global.o:  kernel/global.c  include/type.h  include/const.h
include/protect.h include/proc.h \
          include/global.h include/proto.h
   $(CC) $(CFLAGS) -o $@ $<

kernel/protect.o: kernel/protect.c include/type.h include/const.h
include/protect.h include/proc.h include/proto.h \
          include/global.h
   $(CC) $(CFLAGS) -o $@ $<

kernel/proc.o: kernel/proc.c
   $(CC) $(CFLAGS) -o $@ $<

lib/klib.o:     lib/klib.c     include/type.h     include/const.h
include/protect.h include/string.h include/proc.h include/proto.h \
          include/global.h
   $(CC) $(CFLAGS) -o $@ $<

lib/kliba.o : lib/kliba.asm
   $(ASM) $(ASMKFLAGS) -o $@ $<

lib/string.o : lib/string.asm
   $(ASM) $(ASMKFLAGS) -o $@ $<
```

---

image: realclean everything clean buildimg

最后的产物为 boot.bin loader.bin kernel.bin 其中 kernel.bin 是 elf 格式的

Makefile 里面主要有:
image: realclean everything clean buildimg

realclean: 清空。 rm 中间产物 OBJS + 最后的三个产物

everything: 产生最后的三个产物 boot.bin loader.bin kernel.bin

clean:清空中间产物 OBJS

buildimg:
1.建立一个 img 把 boot.bin 写到 img 的前 512 字节
2.用 mount 指令挂载 img
3.把 loader.bin kernel.bin 加载到挂载 img 所在目录中
4.用 unmount 指令解挂载 img

*这样形成的 a.img（1.44M fat12 格式）
前 512 字节包含了 boot.bin ； 数据区中有 loader.bin 以及 kernel.bin 文件

详细说一下 everything 中三个最后产物到底如何形成的。
首先我们要明确 1）.asm 利用的头文件是.inc
2）.c 利用的头文件是.h
3）.asm 的文件编译都需要 nasm
4） .c 文件的编译都需要 gcc

对于 nasm 以及 gcc 后面都对应的-I 然后一个目录，另外两个参数是源文件和目标文件，中间的目录代表我们编译所需要的头文件从哪里取出。

有三种标识：

A..nasm 后面跟着两种情况.asm，

1.对应 boot 中的，使用-I boot/include
对应文件：boot 中的 boot.asm 以及 loader.asm

2.对应 kernel 或者 lib 中的，使用-I include/ -f elf
对应文件：kernel 中的 kernel.asm syscall.asm lib 中的 kliba.asm string.asm

B..gcc 后面对应一种情况.c，

对应 kernel 或者 lib 中的，使用 -I include/ -c -fno-builtin（加上后面参数的原因是我们所要编译的函数有些跟 gcc 的函数重名，我们不希望利用 gcc 已有的函数，只希望采取 gcc 简单的编译功能，不使用 gcc 本来已经有的与我们同名的函数，所以使用这个参数）
对应文件 kernel 中的 clock.c global.c i8259.c main.c proc.c protect.c start.c

C.在得到中间产物 OBJS 后的产生 kernel.bin 的操作 使用 ld

ld -s -Ttext 0x30400 kernel.bin $OBJS（即中间产物）
对应文件只有 kernel.bin

严格依赖关系中需要有头文件，前面讲到了 asm 头文件都是 inc  c 头文件都是 h
其中，kernel 以及 lib 都是需要编译的 include 里面是头文件，提供给 kernel 以及 lib 里面文件进行编译
另外，boot 文件夹下形成一个自编译的环境，boot.asm 以及 loader.asm 都是需要 boot/include 里面的 inc

三个最后产物 boot.bin 以及 loader.bin 较为简单，一步 nasm 即成功
kernel.bin 比较麻烦，形成-elf 文件，需要把 kernel 以及 lib 中的文件利用 include 里面的头文件全都进行编译以后（Linux 下 gcc 编译结果默认形成 elf 文件，另外.asm 结尾的文件我们利用 nasm -elf 来指明用 nasm 编译生成 elf 文件），再通过 ld -s 进行 kernel.bin 的形成。