使用 GNU Binutils、GCC 和 NASM 编译操作系统

蔡日骏

http://richardtsai.info

2014 年 4 月 29 日

1 介绍

GNU Binutils 中的 GNU Linker 支持 Flat 格式输出，而 NASM 支持 GNU Linker 所需的 ELF 格式输出，因此可以很方便地使用 GNU Linker 把 GCC 编译的目标文件和 NASM 编译的目标文件连接成 Flat 格式的操作系统，这样操作系统的大部分代码都能够使用 C、C++ 等 GCC 支持编译的高级语言编写了。

下面介绍使用 GNU Binutils、GCC 和 NASM 编译操作系统的两种方法，以及一些编写操作系统时需要注意的一些 ABI 方面的问题，最后提供一份自动构建操作系统的 Makefile 样例。

2 使用宿主构建工具链编译

如果你的电脑上已有 GNU Binutils、GCC 和 NASM，那么可以直接使用这套工具链进行编译。这种方式的好处是不需要重新构建工具链。

但是这种方式也有一个问题。GNU Bintuils 和 GCC 的默认参数并不是固定的，而是跟所用版本、工具链构建时指定的目标有关，因此在使用时常常无法确定传入的参数是不是足以覆盖默认参数以生成需要的目标文件。比如，两种常见的 GCC 默认目标是`x86_64-XXX-linux`和`i686-XXX-linux`，而它们默认生成的目标代码的架构是不同的，因此在使用时需要传入不同的参数。

下面以 openSUSE 13.1 默认的 GCC 和 GNU Binutils 为例。GNU Binutils 版本为 `2.23.2`，GCC 版本为`4.8.1`，默认目标为`x86_64-suse-linux`，NASM 版本为`2.10.09`。

2.1 NASM 程序编译

NASM 程序编译比较简单，直接指定输出格式即可。

```
$ nasm -f elf32 -o XXX.o XXX.nasm
```

但是有一个需要注意的问题。NASM 在输出非 Binary 格式时，默认将生成 32 位代码。如果需要得到 16 位实模式下的代码，需要在源文件开头加入下面这个伪指令。

```
BITS 16
```

此外，`ORG` 伪指令将被禁用，程序加载位置的指定将在链接步骤进行。

2.2 C 等程序编译

GCC 编译 C 内核代码例子:

```
$ gcc -c -ffreestanding -m32 -march=i386 -mpreferred-stack-boundary=2 \
    -o kernel.o kernel.c
```

使用 GCC 编译时需要传入下表中的参数。

参数	说明
`-c`	只编译不链接
`-ffreestanding`	使输出程序能独立运行
`-m32`（非`x86_64` 的 GCC 不需要）	生成 32 位代码（16 位也需要此参数，见4.1节）
`-march=i386`	使用 i386 指令集
`-mpreferred-stack-boundary=2`	栈指针按 $2^2 = 4$ 字节对齐
`-fno-exceptions`（非 C++ 不需要）	不使用异常处理
`-fno-rtti`（非 C++ 不需要）	不使用 RTTI
`-o XXX.o`	输出文件名
其他	输入文件名

2.3 链接

链接可以通过直接调用`ld` 命令进行，也可以通过 GCC 进行。下面以`ld` 为例。
例子:

```
$ ld -melf_i386 -ffreestanding -nostdlib -N -Ttext=0x0500 \
    --oformat=binary -o kernel.bin kernel.o utils.o
```

参数解释如下表。

参数	说明
`-melf_i386`	指定输入的目标文件格式
`-ffreestanding`	使输出程序能独立运行
`-nostdlib`	不使用标准库
`-N`	关闭页对齐，但分页内核不能关闭
`-Ttext=0x0500`	指定加载位置，根据需要设置地址
`--oformat=binary`	指定输出格式为 Binary（Flat），根据需要指定
`-o XXX.o`	输出文件名
其他	输入文件名

*注意：某些时候 GCC 会生成依赖 libgcc 的代码，这时链接时需要在**最后**加入 -lgcc 参数，并通过 -L 参数指定 libgcc.a 的位置。*

3 使用交叉构建工具链编译

使用交叉构建工具链需要自己重新编译工具链，但是可移植性更好，编译程序时参数更简洁，不需要考虑版本问题，基本不会出现兼容性问题。

3.1 制作交叉构建工具链

NASM 不存在默认目标的概念，能够直接进行交叉编译，不需要重新编译。但是 GNU Binutils 和 GCC 需要重新编译。

在标准的 POSIX 系统下编译依赖如下：

- G++
- GNU Make
- GNU Bison
- Flex
- GNU GMP
- GNU MPFR
- GNU MPC
- ISL (可选)
- CLooG (可选)

如果在 Mac OS 下编译，可能还需要新版本的 GNU libiconv。

编译前需要进行如下的环境变量设置。

```
export PREFIX=" 安装路径"
export TARGET=i386-elf
export PATH="$PREFIX/bin:$PATH"
```

在 Mac OS 下还需要把CC、CXX、CPP 和LD 这几个环境变量指向 GNU 的工具链 (默认是 LLVM 工具链)。

3.1.1 编译 GNU Binutils

GNU Binutils 源码可以从http://ftp.gnu.org/gnu/binutils/下载。

```
tar -xvf binutils-X.XX.tar.bz2
cd binutils-X.XX
mkdir build
cd build
../configure --target=$TARGET --prefix="$PREFIX" --disable-nls --disable-werror
make
make install # 或 sudo -E make install
```

3.1.2 编译 GCC

GCC 源码可以从http://ftp.gnu.org/gnu/gcc/下载。

```
tar -xvf gcc-X.X.X.tar.bz2
cd gcc-X.X.X
mkdir build
cd build
../configure --target=$TARGET --prefix="$PREFIX" --disable-nls \
    --enable-languages=c,c++ --without-headers
make all-gcc
make all-target-libgcc
make install-gcc # 或 sudo -E make install-gcc
make install-target-libgcc # 或 sudo -E make install-target-libgcc
```

3.2 使用交叉构建工具链编译操作系统

使用上面得到的交叉构建工具链，GCC 编译时只需要传入`-c`、`-ffreestanding`、`-o` 参数即可。LD 链接时不需要`-melf_i386` 参数。

4 13 位实模式 ABI

GCC 本身并无法生成 16 位的指令，因此，使用 GCC 编译能够在实模式下运行的内核需要注意一些 ABI 方面的问题。而保护模式和长模式内核直接编译即可。

4.1 16 位指令前缀

虽然 GCC 无法生成 16 位指令，但是 GNU Binutils 中的 AS 汇编器能够给 32 位指令自动添加 16 位前缀。打开这项功能需要在每个 C 程序源文件开头用内联汇编添加如下一条伪指令：

```
__asm__(".code16gcc\r\n");
```

4.2 调用约定

虽然 16 位指令前缀能使 GCC 生成的指令在实模式下运行，但是其调用约定仍然是 32 位的。

参数传递 参数传递时栈指针至少按 4 字节对齐。默认对齐长度随 GCC 的版本和默认目标不同而不同。在3.1节中得到的交叉构建工具链中的 GCC 默认对齐长度就是 4 字节。

函数返回地址 函数调用时压入栈中的返回地址是 32 位的（高 16 位无效），因此函数返回时直接使用`ret` 指令将会无法返回。需要加入 32 位操作数前缀：

```
o32 ret
```

下面的示例代码是一个 NASM 编写能够在 C 中调用的函数。该函数的作用是修改 IVT 中的指定项。

```
;add_int_handler
;Add a interrupt handler into the IVT
;   0: Interrupt number
;   1: Offset of the handler (must be in the current CS)
;Returns: EAX: The original handler
add_int_handler:
    enter 0, 0
    push bx
    push ds
    xor bx, bx
    mov ds, bx
    mov bl, [bp+6]
    shl bx, 2

    mov ax, cs
    shl eax, 8
    mov ax, [bp+10]

    xchg [ds:bx], eax

    pop ds
    pop bx
    leave
    o32 ret
```

5 Makefile 样例

下面提供一个自动编译、链接、压缩、安装 Multiboot 格式的保护模式内核的 Makefile。该 Makefile 使用一个交叉构建工具链进行编译。压缩后的内核将被安装到`PREFIX/boot` 中，由 Grub 引导启动。

使用时，先运行`make depend` 分析 C 源文件和头文件之间的依赖关系，然后运行`make` 进行编译，最后运行`make install` 进行安装。

```
PREFIX=/media/disk.img
CC=/usr/local/bin/i386-elf-gcc
DEBUGFLAGS=-g3
CFLAGS=-ffreestanding -I./include $(DEBUGFLAGS)
LINKFLAGS=-T link.ld -ffreestanding -nostdlib -lgcc $(DEBUGFLAGS)
NASM=nasm
NASMFLAGS=-f elf
```

```
C_SOURCES=*.c
SRC_DIRS=boot interrupts mm

#Compress the kernel
kernel.gz: kernel.bin
    gzip -c -9 $^ > $@

#Install the kernel
install: kernel.gz
    cp $^ $(PREFIX)/boot

#Link the kernel with GCC using linker script
kernel.bin: kernel.o link.ld submodules
    $(CC) *.o */*.o -o $@ $(LINKFLAGS)

#Build the submodules. Every submodule has its own makefile
submodules:
    for dir in $(SRC_DIRS); do\
        cd $${dir};\
        make;\
        cd ..;\
    done

#Generate the dependencies among .c and .h
depend:
    -for dir in $(SRC_DIRS); do\
        cd $${dir};\
        make depend;\
        cd ..;\
    done
    $(CC) $(C_SOURCES) -MM $(CFLAGS) | while read row; do\
        e_row=$${row//\//\\/};\
        e_row=$${e_row//./\\.};\
        sed -i "s/^$${e_row%:*}.*$$/$${row//\//\\/}/" Makefile;\
        grep -q "$$e_row" Makefile;\
        if [ $$? -eq 1 ]; then\
            echo $${row} >> Makefile;\
        fi;\
    done;

#C programs compile rule
```

```
%.o: %.c
    $(CC) -c $< -o $@ $(CFLAGS)

#NASM programs compile rule
%.o: %.nasm
    $(NASM) $^ -o $@ $(NASMFLAGS)

#Cleanup
clean:
    -rm -rf */*.o
    -rm -rf *.o
    -rm -rf *.bin
    -rm -rf *.gz
```