嵌入式系统原理实验报告四

通信 1503 班 201503090323 叶启彬

实验四 Linux 交叉编译平台实验& Linux 内核编译实验

一、实验目的

1. 理解交叉编译的原理和概念；
2. 掌握在 Linux 下建立交叉编译平台的方法；
3. 掌握使用交叉编译平台编译源代码。
4. 掌握配置和编译 BootLoader（ARMboot）和 Linux 内核的方法；
5. 掌握下载编译好的 BootLoader（ARMboot）和 Linux 内核的方法；
6. 掌握建立 NFS 文件系统的方法。

二、实验设备

1. 硬件：PC 机；ARM9 系统教学实验系统；串口线；网线；服务器。
2. 软件：Linux 操作系统；PC 机操作系统（Windows XP）。

三、实验原理

（一） 交叉编译环境建立的原理

1. 交叉编译是指，在某个主机平台上（比如 PC 上）建立交叉编译环境后，可在其他平台（如 ARM9 实验箱）上运行代码的过程。搭建交叉编译环境，即安装、配置交叉编译工具链。在该环境下编译出嵌入式 Linux 系统所需的操作系统、应用程序等，然后再上传到其他平台（如 ARM9 实验箱）上。

2. 交叉编译工具链是为了编译、链接、处理和调试跨平台体系结构的程序代码。对于交叉开发的工具链来说，在文件名称上加了一个前缀，用来区别本地的工具链。例如，arm_v5t_le 表示是对 arm 的交叉编译工具链；arm_v5t_le_gcc 表示是使用 gcc 的编译器。除了体系结构相关的编译选项以外，其使用方法与 Linux 主机上的 gcc 相同，所以 Linux 编程技术对于嵌入式同样适用。

3. gcc 和 arm-linux-gcc 的区别是什么呢？区别就是 gcc 是 linux 下的 C 语言编译器，编译出来的程序在本地执行，而 arm-linux-gcc 用来在 linux 下跨平台的 C 语言编译器，编译出来的程序在目标机(如 ARM9 实验箱)上执行，嵌入式开发应使用嵌入式交叉编译工具链。

（二） Linux 内核编译的原理

1. 内核配置和编译

内核编译主要分成配置和编译两部分。其中配置是关键，许多问题都是出在配置上。Linux 内核编译配置提供多种方法。如：

```
#make menuconfig  //基于图形工具界面
#make config  //基于文本命令行工具，不推荐使用
#make xconfig  //基于 X11 图形工具界面
```

由于对 Linux 还处在初学阶段，所以选择了简单的配置内核方法，即 make menuconfig。在终端输入 make menuconfig，等待几秒后，终端变成图形化的内核配置界面。进行配置时，大部分选项使用其缺省值，只有一小部分需要根据不同的需要选择硬件介绍。同时内核还提供动态加载的方式，为动态修改内核提供了灵活性。

2. 内核编译系统

Linux 内核的复杂性,使其需要一个强大的工程管理工具。在 Linux 中,提供了 Makefile 机制。Makefile 是整个工程的编译规则。一个工程中源文件不计其数,按其类型、功能、模块被放在不同的目录中。Makefile 定义了一系列的规则来指定哪些文件需要先编译，那些文件需要后编译，哪些文件需要重新编译甚至进行更复杂的操作。Makefile 带来的直接好处就是自动化编译，一当写好，只要一个 make 命令，整个工程自动编译，极大提高效率。

四、实验内容（代码注释及步骤）

（1） Linux 交叉编译平台实验

1. 实验内容

（1）正确运行实验箱；

（2）通过路由器将实验箱和 PC 机连接；

（3）在 Linux 操作系统的服务器上安装交叉编译环境，并编译程序；

（4）在实验箱上运行交叉编译程序结果。

2. 实验步骤：

步骤 0：正确连接实验箱以及 PC 机

将串口连接 PC 机，正确连接实验板电源线，将网线正确连接到实验板。

步骤 1：启动 Linux，使用用户名 student 登录系统

登录服务器：服务器 IP 为 192.168.1.188；登陆的用户为 st2；密码为 123456；

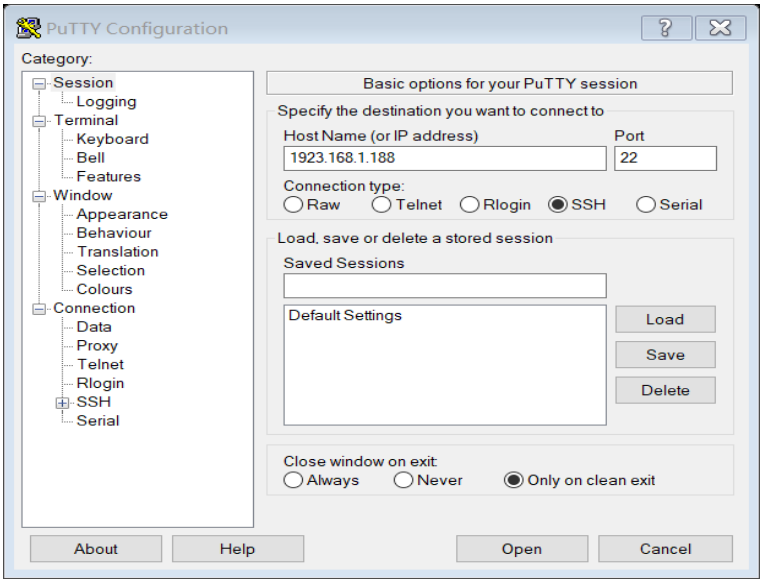

图 1. Putty 登录界面

步骤 2：搭建交叉编译环境

(1) 创建一个文件夹 mv_pro_5.0，进入文件夹 mv_pro_5.0，将/home/shiyan/2018/目录下的软件包 mvltools5_0_0801921_update.tar 复制到当前目录 mv_pro_5.0 下。

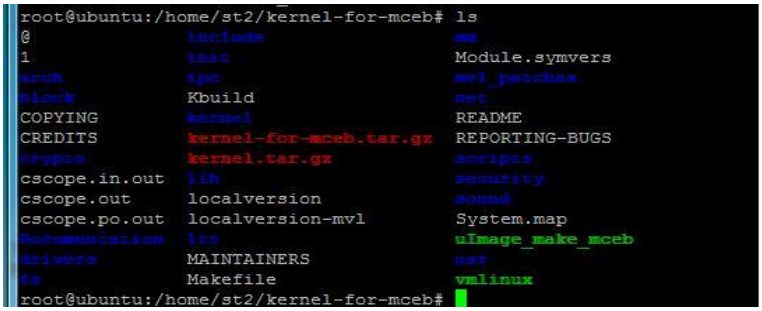

图 2. 找到软件包 mvltools5_0_0801921_update.tar

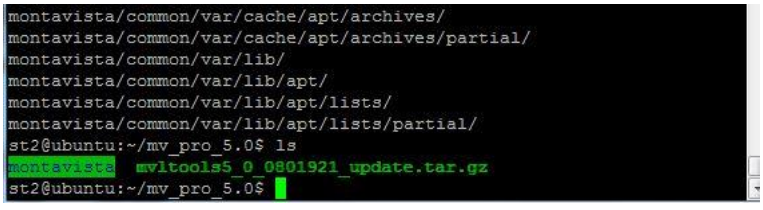

图 3. 将 mvltools5_0_0801921_update.tar 解压到目录 mv_pro_5.0 下

(2) 配置系统环境变量

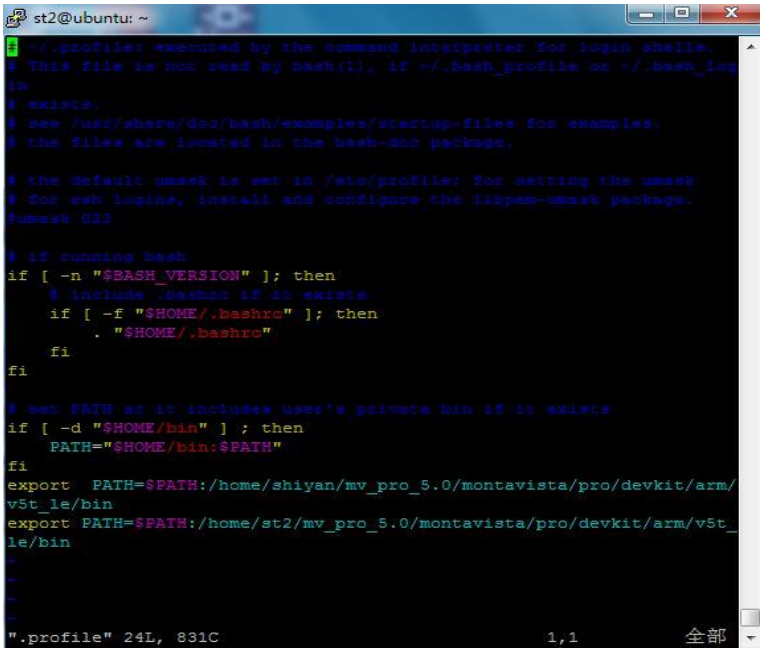

图 4.在.profile 文档最后加上

export PATH=$PATH:/home/st2/mv_pro_5.0/montavista/pro/devkit/arm/v5t_le/bin

(3) 使环境变量生效

输入 source .profile

(4) 检测交叉编译环境是否搭建成功：在命令行中输入 arm_v5t_le-gcc -v。

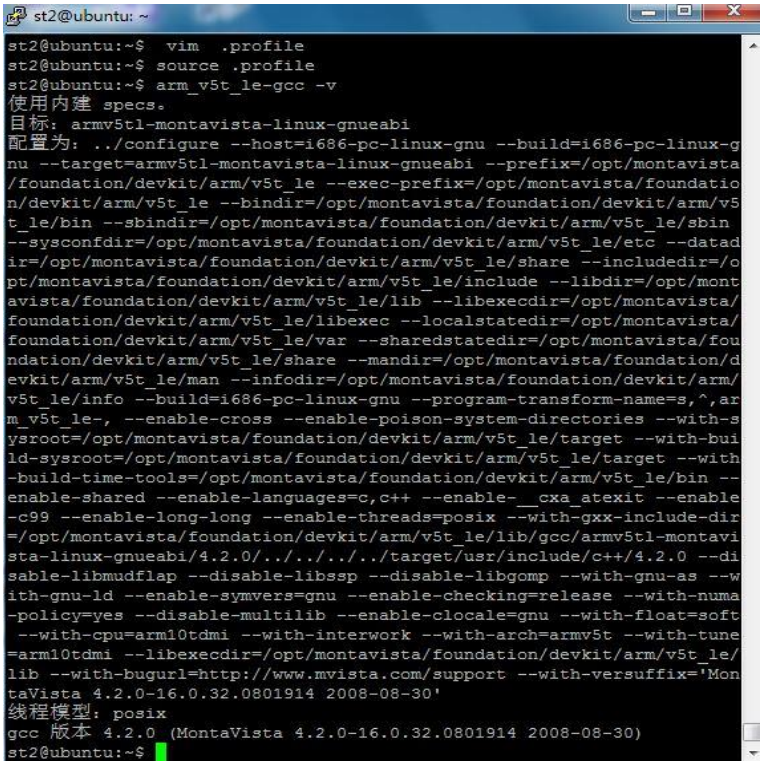

图 5. 交叉编译环境检测

步骤 3：小程序测试

(1) 编写 helloworld1.c 程序：

创建 helloworld1.c 文件：vim helloworld1.c

编写 helloworld1.c 程序

include<stdio.h>

```
int main()
{
    printf("hello world !\n");
    return 0;
}
```

按 Esc 键，再输入:wq!保存退出。

(2) 生成二进制可执行文件 helloworld1：

图 6.生成了可执行二进制文件 helloworld1(在 PC 机上不可以运行)

(3) 正确连接实验箱和 PC 机

将 PC 机与开发板通过 USB 转串口线正确连接，将开发板的电源线网线正确连接，插上电源。

(4)登陆 PuTTY 的 COMX(需要看电脑设备管理器中的串口)，Speed 注意要设为 115200。

(5) 打开设备开关，在 PuTTY 的 COM 口端进行操作，当实验板有打印消息时，按 enter 键使系统停止启动，输入启动参数(上次实验配置过，只要输入即可)。

(6) 通过实验箱运行交叉编译生成的可执行文件:

在 PuTTY 的服务器端进行操作，将生成的二进制可执行文件 helloworld1 由它所在的目录复制到文件系统所在目录（eg: 文件系统所在目录为/home//st2/nfs/filesys_test，就复制到/home//st2/nfs/filesys_test/目录）：sudo cp helloworld1 /home/st2/filesys_test。

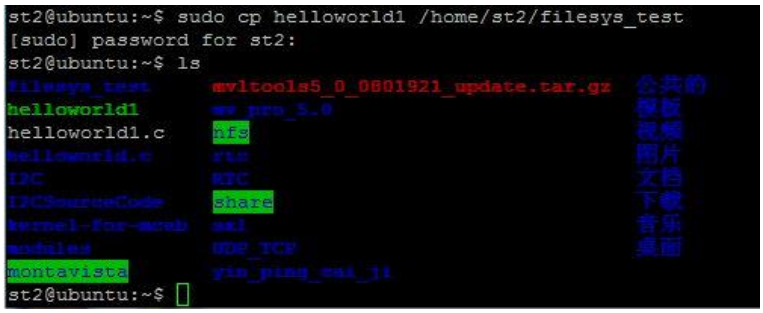

图 7. 将生成的二进制可执行文件 helloworld1 由它所在的目录复制到文件系统所在目录

(7) 运行二进制可执行文件 helloworld1:

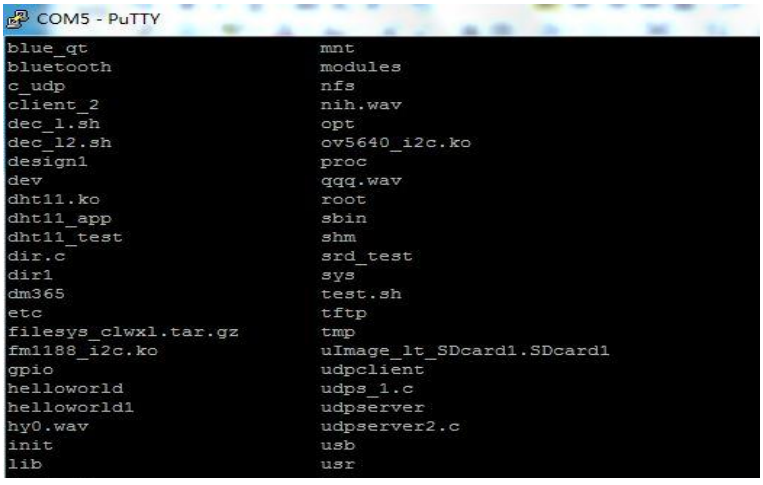

图 8.复制的 helloworld1 二进制可执行文件所在位置

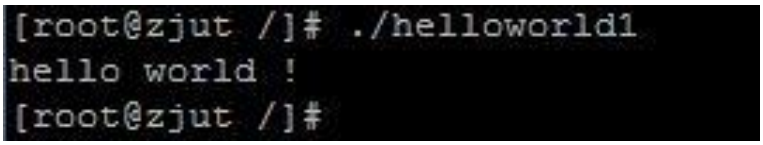

图 9.使用./helloworld1 运行文件，输出 hello world!

（2） Linux 内核编译实验

1. 实验内容

（5）配置和编译 BootLoader（ARMboot）和 Linux 内核的方法；

（6）在实验箱上下载 BootLoader（ARMboot）和 Linux 内核的方法；

（7）在实验箱上配置建立 NFS 文件系统；

（8）在实验箱上移植建立本地文件系统。

2. 实验步骤：

步骤 0：正确连接实验箱以及 PC 机

将串口转 USB 驱动安装正常，实验板电源线正确连接，串口转 USB 线连接至 PC 机。

步骤 1：配置和编译 BootLoader

(1)以用户名登录服务器，进入学生的学习目录，找到内核目录，移动物联网试验箱的内核基于 2.6.18 改进来的 kernel-for-mceb。内核已经事先拷贝好。

(2)进入内核进行配置：

进入内核目录，执行命令为 cd kernel-for-mceb。由于不是第一次编译内核，那么请先运行：sudo make mrproper 清除以前的配置，回到默认配置。

图 10. 清除以前的配置，回到默认配置。

(3) 执行命令为 make menuconfig：

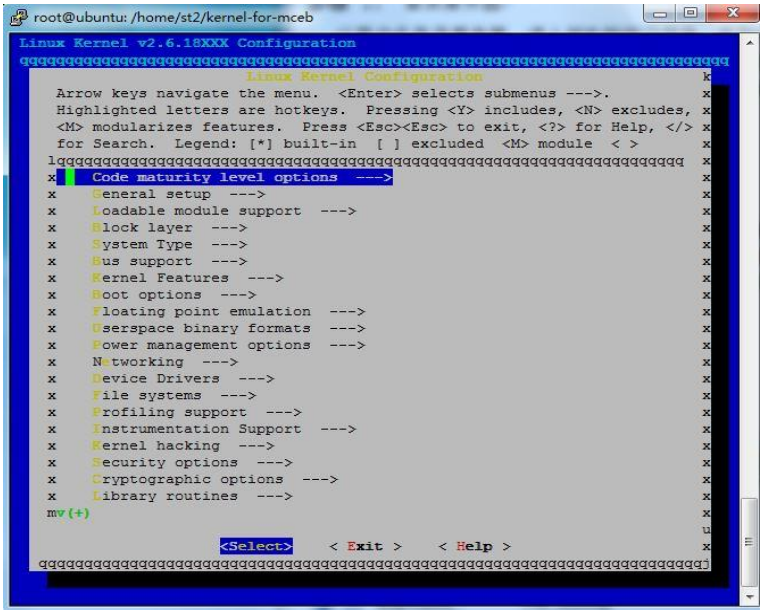

图 11.配置内核界面

编译内核时候往往根据自己的需要来编译自己所需要的。

(4) 在 make menuconfig 命令打开的窗口中选择到 Files systems：

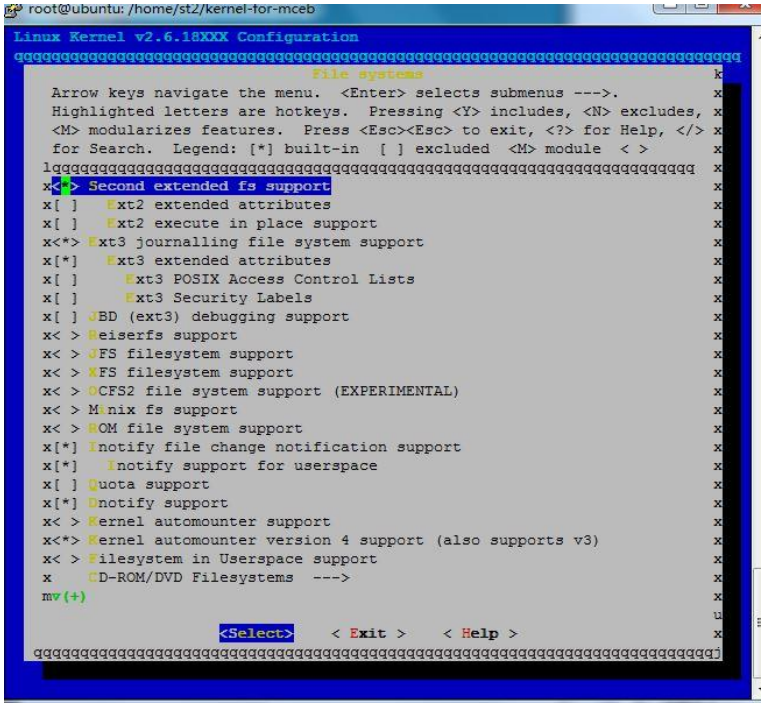

图 12. 进入文件系统配置

(5) 继续选择能支持 ntfs 文件系统类型的选项：

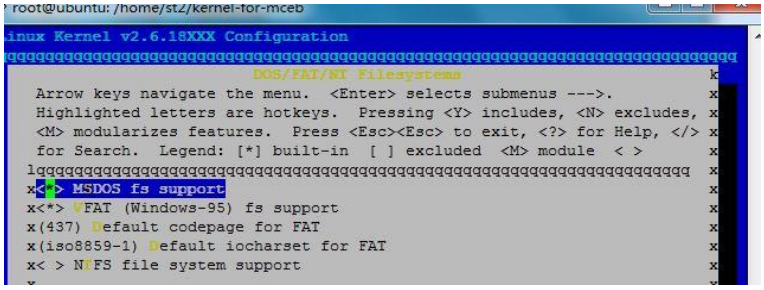

图 13. 进入文件系统选项

(6) 选择我们需要的 ntfs 文件系统类型：

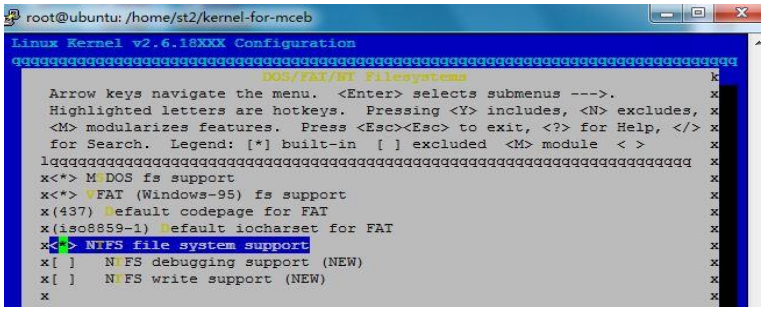

图 14. 选择 ntfs 文件系统选项

(7) 按空格选择编译进内核。并在键盘上按左右键移动光标到退出键按钮，按回车键不断退出：

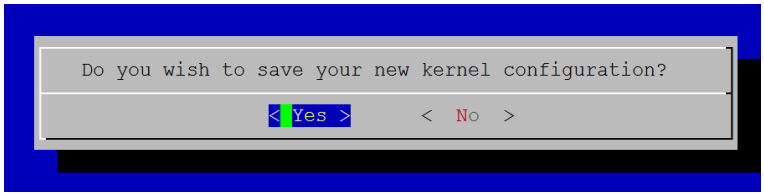

图 15.配置保存界面

步骤 2：配置和编译内核文件系统

在内核目录下使用命令 make uImage。回车键后内核开始编译，等到出现 Image arch/arm/boot/uImage is ready 表示编译结束。使用 exit 退出。

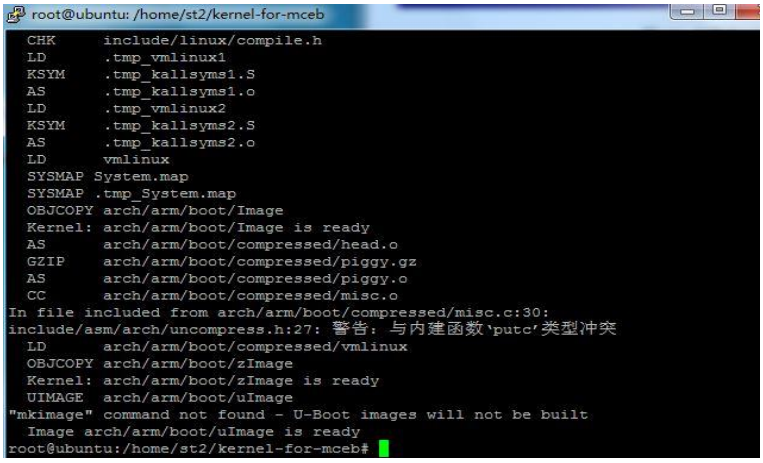

图 16. 编译生成的内核镜像

步骤 3：将 ntfs 文件系统配置成模块方式

按照步骤 1 中的方法，将 NTFS file system support 前面*改变成为 M，即将 ntfs 文件系统配置成为模块加载形式。如下图所示：

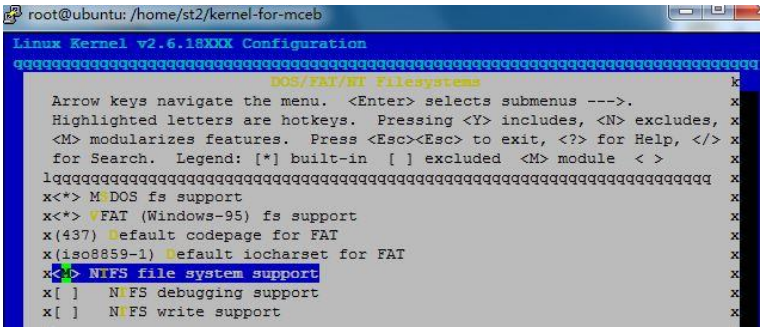

图 17. 内核配置为模块方式

步骤 4：编译模块

执行 make modules 命令，如果不行则使用 sudo su 进入 root 权限使用 source /etc/profile。将会生成一个 ntfs.ko 模块。如下图所示：

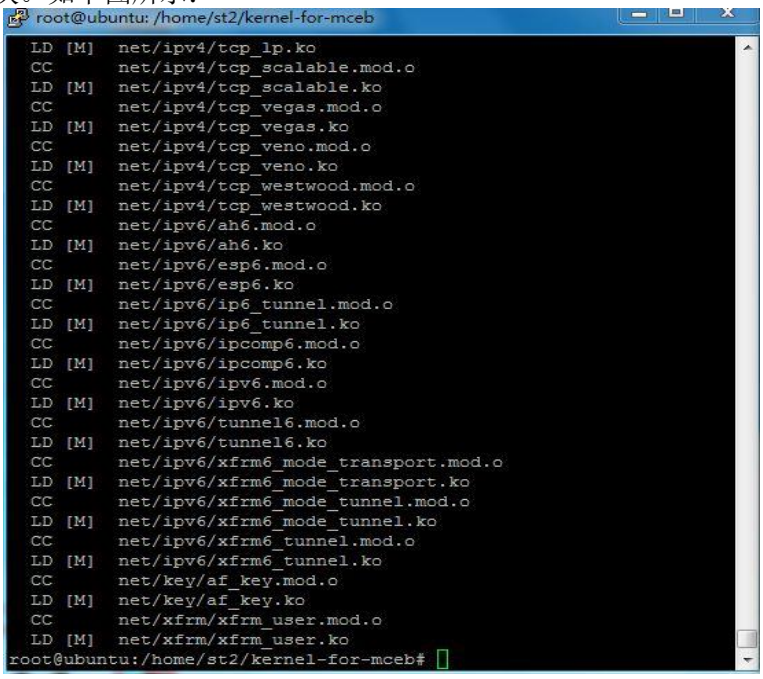

图 18. 编译生成的 ntfs 模块

图 19.查看内核子目录

三、实验总结及心得体会

在本次实验开始前，复习了交叉编译的原理和概念、关于 TCP/IP 网络的相关知识，熟悉了 Linux 操作系统的基本操作；随后在 Linux 系统上建立交叉编译的平台，学会使用交叉编译平台编译源代码。可以清楚地了解到，交叉编译后的二进制文件可以再板子上运行，却不能在服务器上运行。

掌握配置和编译 BootLoader（ARMboot）和 Linux 内核的方法;掌握下载编译好的 BootLoader（ARMboot）和 Linux 内核的方法；掌握建立 NFS 文件系统的方法。

MAKEFILE 文件部分代码注释：

```
#版本信息
VERSION = 2                    #主版本号
PATCHLEVEL = 6                 #次版本号，偶数，稳定版
SUBLEVEL = 18                  #修正版本号
EXTRAVERSION =XXX              #自定义
NAME=Avast! A bilge rat!       #代号

# *DOCUMENTATION*              # *文件*
# To see a list of typical targets execute "make help"
#执行“make help”列出内核编译常用命令列表
# More info can be located in ./README        #更多信息在./README
# Comments in this file are targeted only to the developer, do not
# expect to learn how to build the kernel reading this file.
# Do not print "Entering directory ..."
MAKEFLAGS += --no-print-directory
#不要再屏幕上打印"Entering directory..."，始终被自动的传
#递给所有的下一级 Makefile。
# We are using a recursive build, so we need to do a little thinking
# to get the ordering right.
#我们使用的是递归构建，所以我们需要保证正确的排序。
# Most importantly: sub-Makefiles should only ever modify files in
# their own directory. If in some directory we have a dependency on
# a file in another dir (which doesn't happen often, but it's often
# unavoidable when linking the built-in.o targets which finally
# turn into vmlinux), we will call a sub make in that other dir, and
# after that we are sure that everything which is in that other dir
# is now up to date.
#最重要的是：子目录下的 Makefile 只能在它们所在目录下修改，如果在某个目录中,
我们对另一个目录中的文件有依赖关系(这种情况不会经常发生，但是在链接 built-in.o
的时候通常是不可避免的)，我们将在另一个文件夹内调用子 make，之后保证在那个
文件夹#内的都是最新的
# The only cases where we need to modify files which have global
# effects are thus separated out and done before the recursive
# descending is started. They are now explicitly listed as the
# prepare rule.
#唯一需要修改有全局效应的文件的地方是分离出来在开始递归进入子目录之前。现在
它们被明确列为规则。
# To put more focus on warnings, be less verbose as default
#值得注意的是，使打印信息比默认更不具体
# Use 'make V=1' to see the full commands
#用“make V=1”看完整命令

ifdef V                        #如果 V 被定义
  ifeq ("$(origin V)", "command line")   #如果 V 是通过命令行设置的
    KBUILD_VERBOSE = $(V)      #则 KBUILD_VERBOSE=命令参数
  endif
endif
ifndef KBUILD_VERBOSE          #如果 KBUILD_VERBOSE 没有被定义
  KBUILD_VERBOSE = 0           #则 KBUILD_VERBOSE=0
Endif

# Call checker as part of compilation of C files
#任命一个静态分析工具（默认是 sparse）作为 C 编译器的一部分
# Use 'make C=1' to enable checking (sparse, by default)      #用'make C=1'检查
# Override with 'make C=1 CHECK=checker_executable CHECKFLAGS=....'

ifdef C                        #如果 C 被定义
  ifeq ("$(origin C)", "command line")   #如果 C 是通过命令行设置的
    KBUILD_CHECKSRC = $(C)     #则 KBUILD_ CHECKSRC =命令参数
  endif
endif
ifndef KBUILD_CHECKSRC         #如果 KBUILD_ CHECKSRC 没有被定义
  KBUILD_CHECKSRC = 0          #则 KBUILD_ CHECKSRC =0
endif

# Use make M=dir to specify directory of external module to build
#用 make M=dir 指定编译外部模块的目录
# Old syntax make ... SUBDIRS=$PWD is still supported
#旧的语法 make ... SUBDIRS=$PWD 仍然被支持
# Setting the environment variable KBUILD_EXTMOD take precedence
#优先设置环境变量 KBUILD_EXTMOD

ifdef SUBDIRS                  #如果 SUBDIRS 被定义
  KBUILD_EXTMOD ?= $(SUBDIRS)  #如果 KBUILD_EXTMOD 没有被赋值
endif                          #则 KBUILD_ EXTMOD = SUBDIRS
ifdef M                        #如果 M 被定义
  ifeq ("$(origin M)", "command line")   #如果 M 是通过命令行设置的
    KBUILD_EXTMOD := $(M)      #则 KBUILD_ EXTMOD =命令参数
  endif
endif
# kbuild supports saving output files in a separate directory.
#kbuild 支持把输出文件保存在一个单独目录
# To locate output files in a separate directory two syntaxes are supported.
#有两种方法支持把输出文件保存在一个单独目录
# In both cases the working directory must be the root of the kernel src.
#这两种方法都要求工作目录是内核源码目录
# Use "make O=dir/to/store/output/files/"       # 1) O=用参数 0 设置
# 2) Set KBUILD_OUTPUT
# Set the environment variable KBUILD_OUTPUT to point to the directory
# where the output files shall be placed.
#设置环境变量 KBUILD_OUTPUT 指向存放输出文件的目录
# export KBUILD_OUTPUT=dir/to/store/output/files/
# make          #用 export 声明，使它能传递到下级 makefile
# The O= assignment takes precedence over the KBUILD_OUTPUT environment
#指定参数 0 优先于设置环境变量 KBUILD_OUTPUT
# variable.     #变量值参数

# KBUILD_SRC 放在 OBJ 目录，KBUILD_SRC 不打算给用户用
# KBUILD_SRC is set on invocation of make in OBJ directory
# KBUILD_SRC is not intended to be used by the regular user (for now)
ifeq ($(KBUILD_SRC),)
#在内核源码文件夹内调用 make
# OK, Make called in directory where kernel src resides
#想不想把输出文件放在不同文件夹内？
# Do we want to locate output files in a separate directory?
#如果 0 被定义，如果 0 是通过命令行设置的，则 KBUILD_ OUTPUT =命令参数
```

```
ifdef O
  ifeq ("$(origin O)", "command line")
    KBUILD_OUTPUT := $(O)
  endif
endif

# That's our default target when none is given on the command line
#如果命令没有指定，默认就是_all
PHONY := _all
_all:

ifneq ($(KBUILD_OUTPUT),)
#如果设定输出目录
# Invoke a second make in the output directory, passing relevant variables
#调用目录的第二个 make，传递相关变量
# check that the output directory actually exists
#检查输出目录确实存在
saved-output := $(KBUILD_OUTPUT)
#备份输出目录
KBUILD_OUTPUT := $(shell cd $(KBUILD_OUTPUT) && /bin/pwd)
#进入输出目录，用 pwd 命令将输出目录重新赋值给变量 KBUILD_OUTPUT
#根据重新赋值后的 KBUILD_OUTPUT 判断输出目录是否存在

$(if $(KBUILD_OUTPUT),, \                    #存在，则什么也不做
     $(error output directory "$(saved-output)" does not exist)) #不存在则报错
PHONY += $(MAKECMDGOALS)
#定义两个为目标：一个是除去_all 后的$(MAKECMDGOALS)，另一个是_all
$(filter-out _all,$(MAKECMDGOALS)) _all:
#如果 KBUILD_VERBOSE 为 1 则等于空，如果$(KBUILD_VERBOSE:1=)不为空则等于$@，跳#转到输出目录，执行
make$(if $(KBUILD_VERBOSE:1=),@)$(MAKE) -C $(KBUILD_OUTPUT) \
#将 CURDIR（当前工作路径）赋值给 KBUILD_SRC
    KBUILD_SRC=$(CURDIR) \
    KBUILD_EXTMOD="$(KBUILD_EXTMOD)" -f $(CURDIR)/Makefile $@
    #赋值 KBUILD_EXTMOD

# Leave processing to above invocation of make          #结束子 make，返回上级 make
skip-makefile := 1
endif # ifneq ($(KBUILD_OUTPUT),)
endif # ifeq ($(KBUILD_SRC),)
# We process the rest of the Makefile if this is the final invocation of make
#如果正在运行最外层的 make，则处理 makefile 的剩余部分

ifeq ($(skip-makefile),)                    #判断是否是最外层的 make
# If building an external module we do not care about the all: rule
#如果正在编译动态加载模块，则不关心 all
# but instead _all depend on modules
#但是_all 依赖于模块

PHONY += all                                #声明为目标 all
ifeq ($(KBUILD_EXTMOD),)                    #判断是否编译动态加载模块
_all: all                                   #是，则_all 依赖于 all
else
_all: modules                               #否，则依赖于模块
endif

#看 KBUILD_SRC 是否为空，设置源码目录
srctree         := $(if $(KBUILD_SRC),$(KBUILD_SRC),$(CURDIR))
TOPDIR          := $(srctree)               #顶层目录
# FIXME - TOPDIR is obsolete, use srctree/objtree
objtree         := $(CURDIR)                # CURDIR 为 make 的默认环境变量
src             := $(srctree)               #设置源文件的目录为当前目录
obj             := $(objtree)               #设置目标文件的输出目录为当前目录
#添加搜索路径
VPATH           := $(srctree)$(if $(KBUILD_EXTMOD),:$(KBUILD_EXTMOD))
export srctree objtree VPATH TOPDIR
#传递 srctree objtree VPATH TOPDIR 到下级 makefile

# SUBARCH tells the usermode build what the underlying arch is.    That is set
# first, and if a usermode build is happening, the "ARCH=um" on the command
# line overrides the setting of ARCH below.    If a native build is happening,
# then ARCH is assigned, getting whatever value it gets normally, and
# SUBARCH is subsequently ignored.
# uname –m 显示系统类型    sed 文本编辑    -e s/i.86/i386/  用 i386 替换 i.86
SUBARCH := $(shell uname -m | sed -e s/i.86/i386/ -e s/sun4u/sparc64/ \
                                -e s/arm.*/arm/ -e s/sa110/arm/ \
                                -e s/s390x/s390/ -e s/parisc64/parisc/ \
                                -e s/ppc.*/powerpc/ -e s/mips.*/mips/ )

#交叉编译和为 gcc、bin 工具选择不同设置
# Cross compiling and selecting different set of gcc/bin-utils
# ---------------------------------------------------------------------------
# When performing cross compilation for other architectures ARCH shall be set
# to the target architecture. (See arch/* for the possibilities).
#当交叉编译其他体系内核时，ARCH 应该设置为目标体系（看母驴 arch/#有哪些体系支持）
# ARCH can be set during invocation of make: #ARCH 可以在 make 执行间设置
# make ARCH=ia64
# Another way is to have ARCH set in the environment.
# The default ARCH is the host where make is executed.
#另一种方法是设置 ARCH 环境变量，默认是当前执行的宿主机
# CROSS_COMPILE specify the prefix used for all executables used
# during compilation. Only gcc and related bin-utils executables
# are prefixed with $(CROSS_COMPILE).
# CROSS_COMPILE 作为编译时的前缀，只有 gcc 和相关的 bin-utils 使用$(CROSS_COMPILE)前缀
# CROSS_COMPILE can be set on the command line
#CROSS_COMPILE 可以在命令行设置，也可以在环境变量中设置
# make CROSS_COMPILE=ia64-linux-
# Alternatively CROSS_COMPILE can be set in the environment.
# Default value for CROSS_COMPILE is not to prefix executables
#默认 CROSS_COMPILE 为空
# Note: Some architectures assign CROSS_COMPILE in their arch/*/Makefile
#注意：一些体系的 CROSS_COMPILE 可以在他们的 arch/*/Makefile 中设置

#ARCH := $(shell if [ -f .mvl_target_cpu ]; then \cat .mvl_target_cpu; \
#       else \echo $(SUBARCH); \fi)
#CROSS_COMPILE      = $(shell if [ -f .mvl_cross_compile ]; then \
#                           cat .mvl_cross_compile; \fi)
ARCH = arm
CROSS_COMPILE = arm_v5t_le-
# Architecture as present in compile.h
UTS_MACHINE := $(ARCH)

KCONFIG_CONFIG    ?= .config              #生成配置文件
# SHELL used by kbuild      这里 shell 中的 if [ -x file ]测试 file 是否可执行
CONFIG_SHELL := $(shell if [ -x "$$BASH" ]; then echo $$BASH; \
          else if [ -x /bin/bash ]; then echo /bin/bash; \
          else echo sh; fi ; fi)

HOSTINCLUDE    = $(shell if [ -n "`which mvl-whereami 2> /dev/null`" ] ; then \
                        echo "-I`mvl-whereami`/../include"; \fi)
HOSTCC          = gcc
HOSTCXX         = g++
HOSTCFLAGS      =  -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer $(HOSTINCLUDE)
HOSTCXXFLAGS = -O2

# Decide whether to build built-in, modular, or both.
# Normally, just do built-in.

KBUILD_MODULES :=
KBUILD_BUILTIN := 1
#       If we have only "make modules", don't compile built-in objects.
#       When we're building modules with modversions, we need to consider
#       the built-in objects during the descend as well, in order to
#       make sure the checksums are up to date before we record them.

#如果执行”make modules”则这里在这里重新处理 KBUILD_BUILTIN :=1
ifeq ($(MAKECMDGOALS),modules)
  KBUILD_BUILTIN := $(if $(CONFIG_MODVERSIONS),1)
endif

#       If we have "make <whatever> modules", compile modules
#       in addition to whatever we do anyway.
#       Just "make" or "make all" shall build modules as well
#如果执行"make all","make _all","make modules","make"中任一个命令则在这里重新处理 KBUILD_MODULES
ifneq ($(filter all _all modules,$(MAKECMDGOALS)),) ) #filter 过滤出指定的字符串
  KBUILD_MODULES := 1                   #这里表示如果不为空则编译模块
endif

ifeq ($(MAKECMDGOALS),)
  KBUILD_MODULES := 1
endif

#导出变量给子 make
export KBUILD_MODULES KBUILD_BUILTIN
export KBUILD_CHECKSRC KBUILD_SRC KBUILD_EXTMOD
```