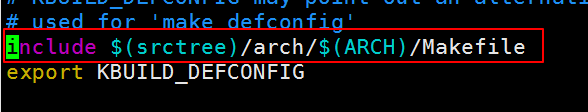
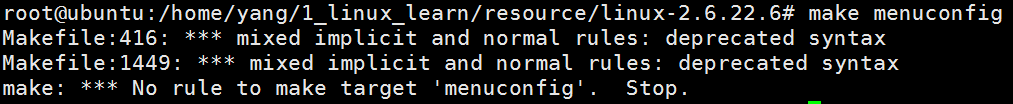
make uImge -----> uImge的目标是在arch/arm/Makefile中。所以arch/arm/Makefile应该被包含在顶层目录中。



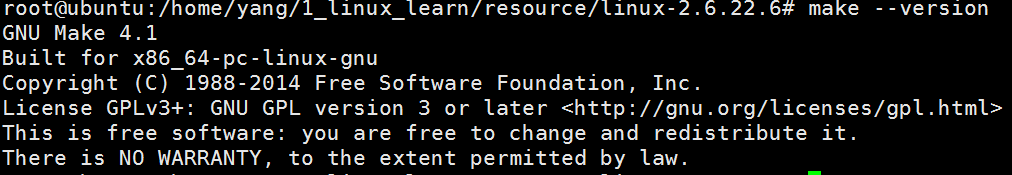
auto.conf也会被包含在顶层Makefile中



Make工具的版本可能不兼容，会出现如下的结果：



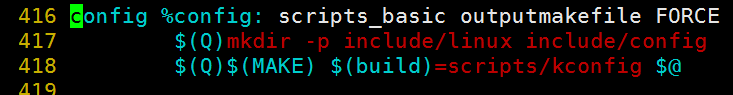
Linux2.6的源码可能是根据某个版本的make工具所支持的语法来编写的，但是由于后面的make工具版本太低就可能出现某些内核可能编译不过。

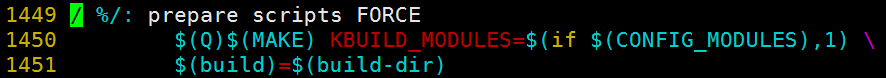


由于我的make 版本高于Linux2.6使用的makefile支持的make版本的语法，所以要修改一下Makefile的文件或者是降低make工具的版本就可以了，至于怎么知道相应内核使用的make工具的版本，这个我现在还不知道，这个可以在网上查。

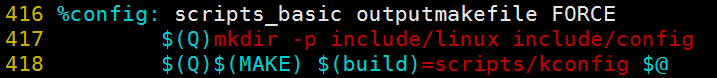
如在416和1449行，报错：

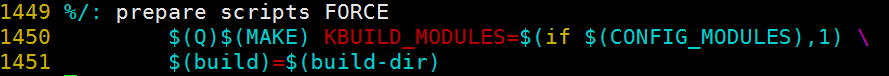
混合隐式和普通规则：不推荐使用的语法。所以先看416行和1449行



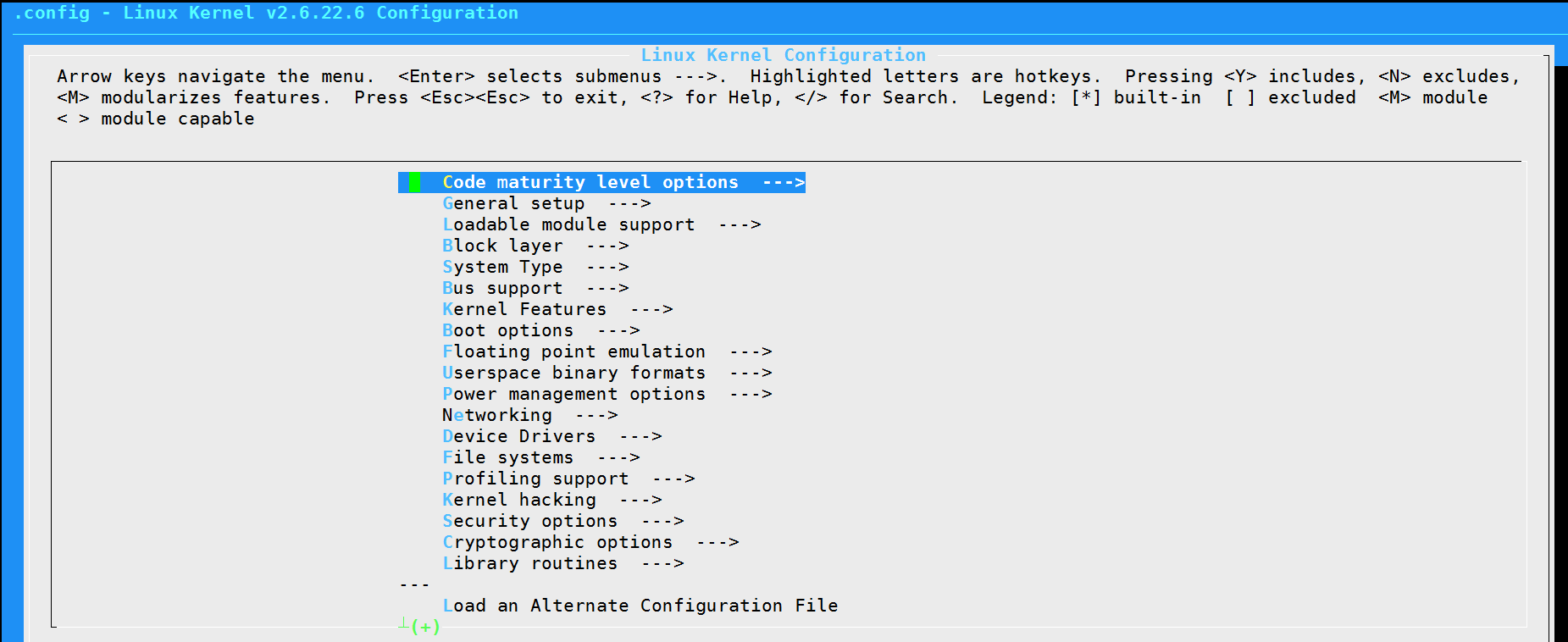


修改后：



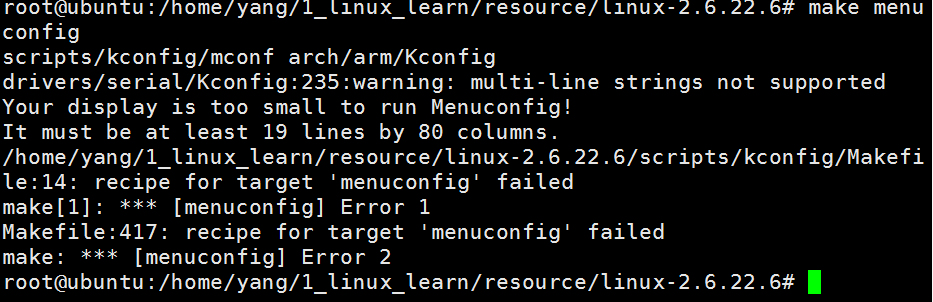


这下执行make menuconfig就会出现熟悉的画面了



经过make menuconfig后最终会生成一个.config的文件

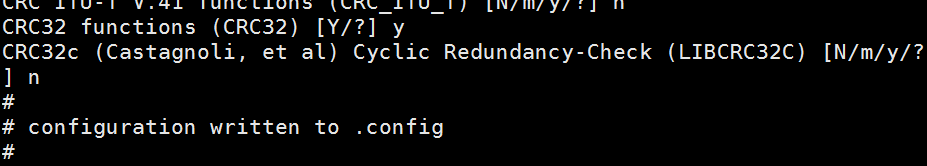
(注意: 如果用的是Xshell则，Xshell托动的窗口太小会出现如下图所示的错误，全屏就是可以的。)



一般配置有3种方式：

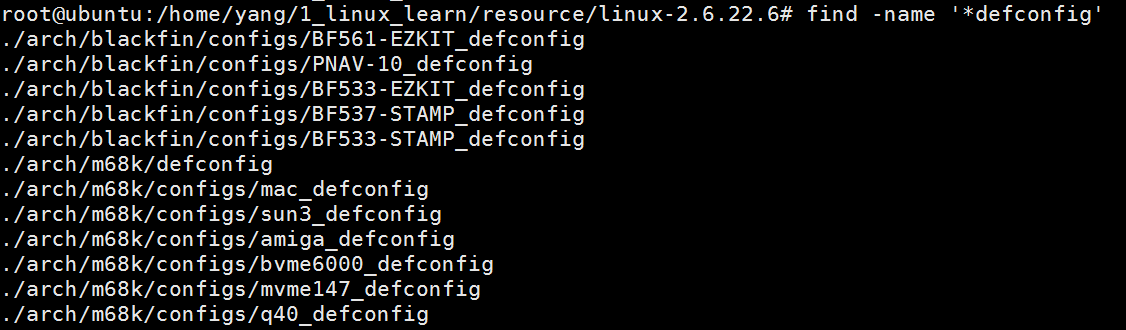
1. 直接根据make menuconfig在成千上万个配置项中选则；
2. 根据默认的xxx\_defconfig(如果是arm,则文件在arch/arm/configs中找相似的配置文件)并在其中进行修改；
3. 厂家会提供一个.config

如果是用2方式，则在最顶层目录下,执行make xxx\_defconfig,最终会生成一个.confg的文件



然后再执行make menuconfig，就会出现菜单文件。

一般是用户就可以使用3方式，如果是厂家的话，一般是用2方式。



Vim .config

Xxx=y或=m,或等于某些数值,=1则编译到内核，=m是编译到模块。



如CONFIG\_DM9000=y,则DM9000会被编译到内核，如果CONFIG\_DM9000=m则会被编译成模块。

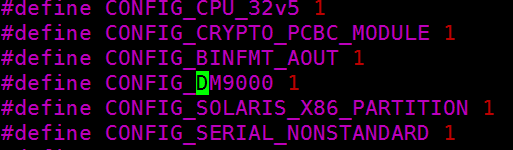
Grep –Ren –color=auto ‘CONFIG\_DM9000’

查看CONFIG\_DM9000在那些文件包含CONFIG\_DM9000

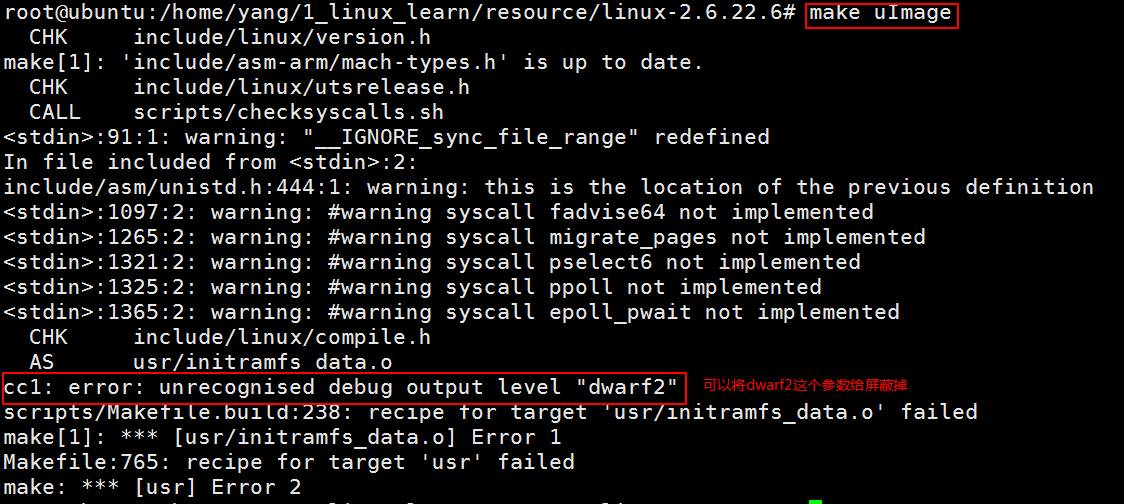
1. C源码(宏，来源于autoconf.h)
2. 子目录的Makefile
3. Include/config/auto.conf(会被顶层Makefile包含)
4. Include/linux/autoconf.h(自动生成，其中的内容来源于.confg)

当执行了make uImage 后会自动生成auto.conf 和autoconf.h

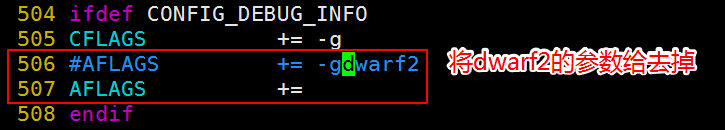
打开autoconf.h如果CONFIG\_DM9000=y或者CONFIG\_DM9000=m, 则头文件中CONFIG\_DM9000都会等于1，C源码中会用到这个宏。



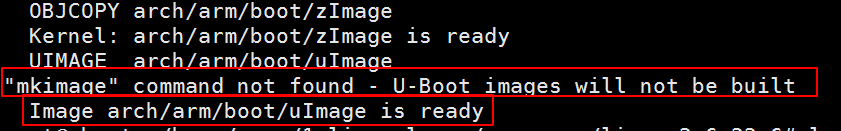
当配置好后，进行make uImage,如果这步报如下错误



在顶层目录中将dwarf2的这个参数给去掉就OK了，然后执行make uImage。

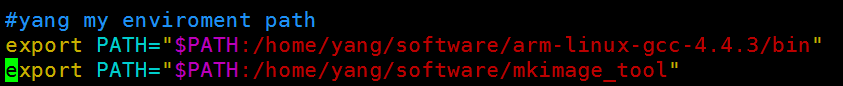


当执行make uImage后可能不会产生uImage的文件，可能是没有mkimage的文件，因为这个文件是编译uboot时产生的。只需要将uboot产生的mkimage可执行文件的环境变量配置好就可以了。



编辑vim ~/.bashrc

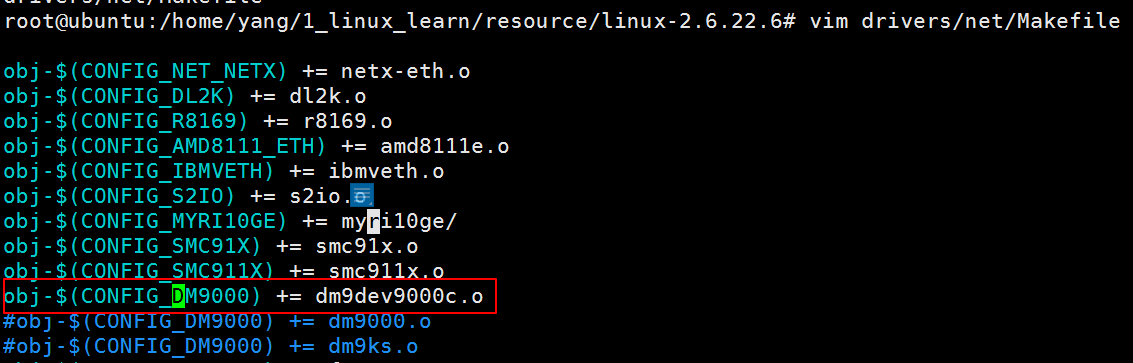
将相应的环境变量添加到文件中，执行source ~/.bashrc 使得环境变量生效。



当添加环境变量后，再执行make uImage后，就会在arch/arm/boot/下会有一个uImage的文件，我们的需要的内核文件就是这个。

子目录下的Makefile中obj\_y += xxx.o 是编译进内核的，obj\_m += yyy.o 是编译进模块的，最终被编译成yyy.ko。

如果在.config中定义了CONFIG\_DM9000=y,则dm9dev9000c.c会被编译进内核，CONFIG\_DM9000=m则会被编译成模块文件(ko)



多个文件被编译成一个模块

obj-m += ab.o

ab-objs := a.o b.o

a.c -> a.o

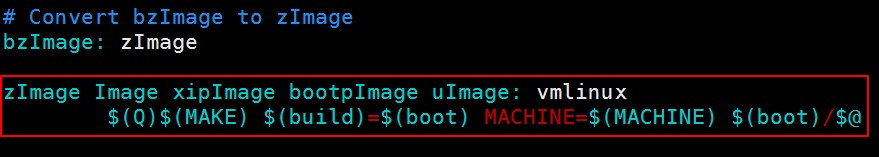
🡺 ab.ko

b.c-> b.o

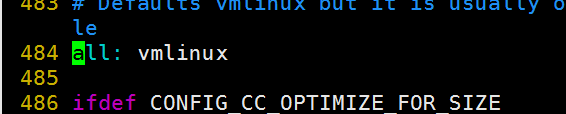
分析内核的Makefile

Make uImage依赖vmlinux

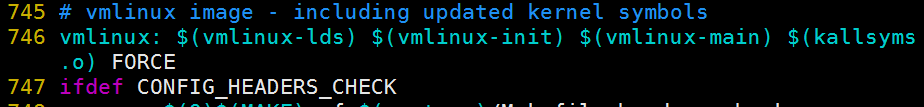
在arch/arm/Makefile中可以看到uImage依赖于vmliux, uImage = 头部 + vmlinux(真正的内核)



顶层Makefile,也依赖vmlinux



Vmlinux的依赖



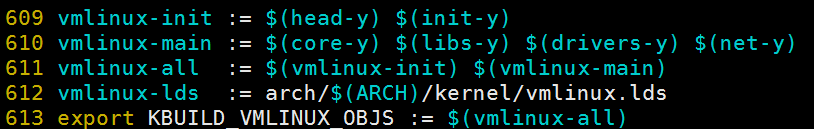
vmlinux: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) $(kallsyms .o) FORCE

vmlinux-lds 链接脚本

vmlinux-init 一些初始化函数

vmlinux-main 一些主要的函数

顶层Makefile中



vmlinux-init := $(head-y) $(init-y)

vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)

vmlinux-all := $(vmlinux-init) $(vmlinux-main)

vmlinux-lds := arch/$(ARCH)/kernel/vmlinux.lds

在arch/arm/Makefile中：

head-y := arch/arm/kernel/head$(MMUEXT).o arch/arm/kernel/init\_task.o

在顶层Makefile中

init-y := init/

init-y := $(patsubst %/, %/built-in.o, $(init-y)) 这是一个函数，最终等于init/build-in.o

在init目录下所有涉及的文件会被编译成build-in.o文件

core-y := usr/

ifeq ($(KBUILD\_EXTMOD),)

core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/

core-y := $(patsubst %/, %/built-in.o, $(core-y))

所有的目录会被编译成build-in.o，如：usr/built-in.o kernel/built-in.o mm/built-in.o fs/built-in.o ipc/built-in.o security/built-in.o crypto/built-in.o block/built-in.o

libs-y := lib/

libs-y := $(libs-y1) $(libs-y2)

编译成lib/lib.a lib/built-in.o

libs-y1 := $(patsubst %/, %/lib.a, $(libs-y))

libs-y2 := $(patsubst %/, %/built-in.o, $(libs-y))

drivers-y := drivers/ sound/

drivers-y := $(patsubst %/, %/built-in.o, $(drivers-y))

编译成drivers/built-in.o sound/built-in.o

net-y := net/

net-y := $(patsubst %/, %/built-in.o, $(net-y))

编译成net/built-in.o

原材料就是上面的文件，看最终的链接命令，make uImage V=1

(V=1 看详细信息)

(链接生成vmlinux)

arm-linux-ld -EL -p --no-undefined -X -o vmlinux

-T arch/arm/kernel/vmlinux.lds

arch/arm/kernel/head.o arch/arm/kernel/init\_task.o init/built-in.o

--start-group usr/built-in.o arch/arm/kernel/built-in.o arch/arm/mm/built-in.o arch/arm/common/built-in.o arch/arm/mach-s3c2410/built-in.o arch/arm/mach-s3c2400/built-in.o arch/arm/mach-s3c2412/built-in.o arch/arm/mach-s3c2440/built-in.o arch/arm/mach-s3c2442/built-in.o arch/arm/mach-s3c2443/built-in.o arch/arm/nwfpe/built-in.o arch/arm/plat-s3c24xx/built-in.o kernel/built-in.o mm/built-in.o fs/built-in.o ipc/built-in.o security/built-in.o crypto/built-in.o block/built-in.o arch/arm/lib/lib.a lib/lib.a arch/arm/lib/built-in.o lib/built-in.o drivers/built-in.o sound/built-in.o net/built-in.o --end-group .tmp\_kallsyms2.o

分析Makefile的目的：

1. 确定第1个文件：arch/arm/kernel/head.S
2. 链接脚本：arch/arm/kernel/vmlinux.lds(如果链接脚本中没有.o文件，则根据链接时.o文件的顺序来判断，第1个则是head.o,由第head.o可以推出源文件是head.S或者是head.c,显然是head.S)

在uboot中：

theKernel (0, bd->bi\_arch\_number, bd->bi\_boot\_params);

bd->bi\_arch\_number 机器ID

bd->bi\_boot\_params 参数所存储的位置

内核肯定是要处理这些参数.

前文中知道了，内核启动的第1个文件，head.S 在source insight 中收索head.S可以发现，两个head.S分别在arch/arm/boot/compressed和arch/arm/kernel中。在compressed目录中的head.S中有自解压的代码。(自解压代码 + 压缩后的内核) . 一般没有压缩的内核，看在kernel中的head.S

ENTRY(stext)

msr cpsr\_c, #PSR\_F\_BIT | PSR\_I\_BIT | SVC\_MODE @ ensure svc mode

@ and irqs disabled

/\* 读处理ID,在\_\_lookup\_processor\_type中查看读取的ID, 当前内核是否支持该机器ID \*/

mrc p15, 0, r9, c0, c0 @ get processor id

bl \_\_lookup\_processor\_type @ r5=procinfo r9=cpuid

movs r10, r5 @ invalid processor (r5=0)?

beq \_\_error\_p @ yes, error 'p'

/\* 机器ID的判断，一个内核支持那些单板都是程序定下来的 \*/

bl \_\_lookup\_machine\_type @ r5=machinfo

movs r8, r5 @ invalid machine (r5=0)?

beq \_\_error\_a @ yes, error 'a'

bl \_\_create\_page\_tables

注：机器ID是在board\_init中定义

#define MACH\_TYPE\_TOTO 361

#define MACH\_TYPE\_S3C2440 362

#define MACH\_TYPE\_KS8695P 363

#define MACH\_TYPE\_SE4000 364

\_\_arch\_info\_begin = .;

\*(.arch.info.init)

\_\_arch\_info\_end = .;

#define MACHINE\_START(\_type,\_name) \

static const struct machine\_desc \_\_mach\_desc\_##\_type \

\_\_used \

\_\_attribute\_\_((\_\_section\_\_(".arch.info.init"))) = { \

.nr = MACH\_TYPE\_##\_type, \

.name = \_name,

#define MACHINE\_END \

};

/\* 如果内核将此编译进去后，则内核就支持了，该机器ID \*/

MACHINE\_START(SMDK2410, "SMDK2410") /\* @TODO: request a new identifier and switch

\* to SMDK2410 \*/

/\* Maintainer: Jonas Dietsche \*/

.phys\_io = S3C2410\_PA\_UART,

.io\_pg\_offst = (((u32)S3C24XX\_VA\_UART) >> 18) & 0xfffc,

/\* S3C2410\_SDRAM\_PA 是0x30000000,即刚好是0x30000100 \*/

.boot\_params = S3C2410\_SDRAM\_PA + 0x100,

.map\_io = smdk2410\_map\_io,

.init\_irq = s3c24xx\_init\_irq,

.init\_machine = smdk2410\_init,

.timer = &s3c24xx\_timer,

MACHINE\_END

\_\_lookup\_machine\_type:

adr r3, 3b @r3 就等于3b的地址

ldmia r3, {r4, r5, r6} @r4 = . 3: .long .这个地址的虚拟地址 r5 = \_\_arch\_info\_begin, r6 = \_\_arch\_info\_end

sub r3, r3, r4 @ get offset between virt&phys，计算虚拟地址与物理地址的偏移

add r5, r5, r3 @ convert virt addresses to

add r6, r6, r3 @ physical address space

1: ldr r3, [r5, #MACHINFO\_TYPE] @ get machine type

teq r3, r1 @ matches loader number?

beq 2f @ found

add r5, r5, #SIZEOF\_MACHINE\_DESC @ next machine\_desc

cmp r5, r6

blo 1b

mov r5, #0 @ unknown machine

2: mov pc, lr

Start\_kernel

Setup\_arch //解析u-boot传入启动参数

Setup\_command\_line //解析u-boot传入启动参数

Reset\_init

Kernel\_init

prepare\_namespace

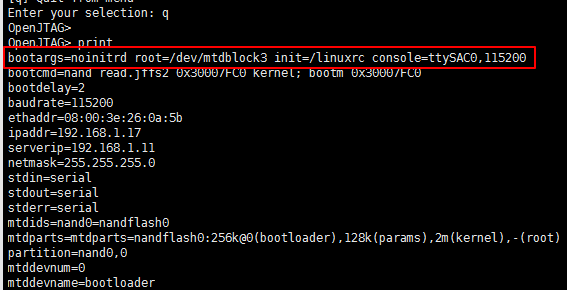
mount\_root //挂接根文件系统

init\_post

// 执行应用程序

解析root=xxx,的参数。





1. Uboot: 启动内核

2、内核：启动应用程序(应用程序位于那里，在根文件系统中)

3、构建根文件系统

内核怎样启动第1个应用程序：

文件系统

执行 ls –l /bin/ls 会显示链接到busybox,

Ls 和 busybox ls 是一样的效果

在内核中

run\_init\_process("/sbin/init");

run\_init\_process("/etc/init");

run\_init\_process("/bin/init");

run\_init\_process("/bin/sh");

/sbin/init 进程也是链接到busybox,所以要看/sbin/init中做了那些事情就要看busybox的源码。

signal(SIGHUP, exec\_signal);

signal(SIGQUIT, exec\_signal);

signal(SIGUSR1, shutdown\_signal);

signal(SIGUSR2, shutdown\_signal);

signal(SIGINT, ctrlaltdel\_signal);

signal(SIGTERM, shutdown\_signal);

signal(SIGCONT, cont\_handler);

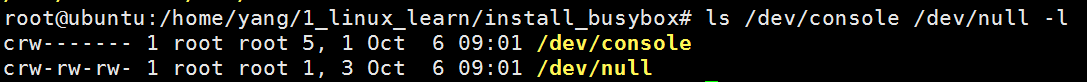
signal(SIGSTOP, stop\_handler);

signal(SIGTSTP, stop\_handler);

如：当按下ctrl + alt + del的时候内核会给init进程发一个信号(SIGINT),当接收到信号后就会执行ctrlaltdel\_signal处理函数🡪 run\_actions(CTRLALTDEL);会执行这一类的程序。

Init进程需要那些东西：

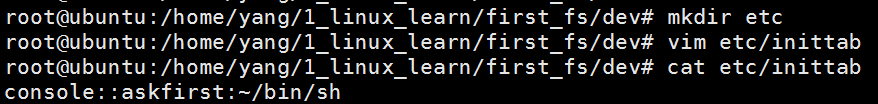
1. /dev/console 打开 /dev/null 如果没有设置配置文件中的ID,则标准输出，标准输入，标准错误，就会定位到/dev/null
2. /etc/inittab 配置文件
3. 配置文件里指定的应用程序
4. C库(应用程序使用的fopen, fread等)
5. Init在busybox中，所以需要busybox



c 字符设备，5 主设备号, 1次设备号 /dev/console



创建两个设备文件，按照当前ubuntu的设备文件的主设备号和次设备号创建。



创建一个最简单的inittab

~/bin/sh程序的标准输出，标准输入，标准错误定位到console

文件系统的格式，yaffs是针对nandflash每页是512K的，yaffs2是针对NANDFLASH每页是2K的。

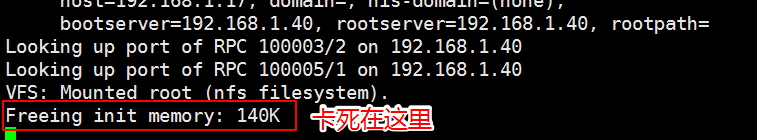
文件系统容易出现的问题：

1. C库版本不匹配(编译busybox用的a编译器，但是却从b编译器中拷贝的C库)



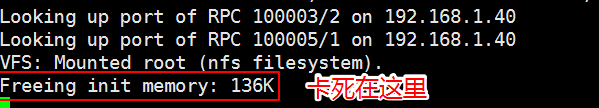
Kernel panic - not syncing: No init found. Try passing init= option to kernel

1. Busybox用arm-linux-gcc 4.4.3 编译，同样的库也用的是4.4.3交叉工具链的。但内核还是用的arm-linux-gcc 3.4.5来编译。



Freeing init memory: 140K

3．Busybox用arm-linux-gcc 4.4.3 编译，同样的库也用的是4.4.3交叉工具链的，内核也用4.4.3来编译（即使将u-boot用4.4.3编译也是一样的效果，当然u-boot和内核是单向的不用保持编译器一致）。



Freeing init memory: 136K

4 . Busybox和C库都用 3.4.5交叉工具链，就能正常的启动内核。不和道是不是busybox要配相应的编译器才能正常的用哦。

内核提供一个proc的虚拟文件系统，用来收集当前有那些文件在跑,挂载proc文件系统后就可以执行ps命令显示当前系统正在执行的程序有那些了。

挂载proc文件系统

mount –t proc none /proc

mount -t type device dir

The proc filesystem is not associated with a special

device, and when mounting it, an arbitrary keyword,

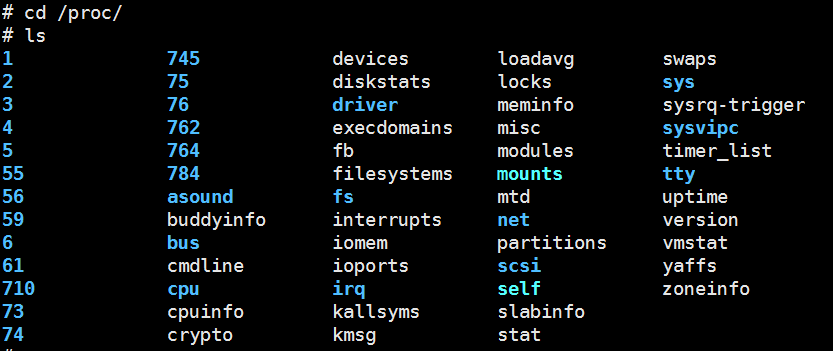
such as proc can be used instead of a device specifica‐

tion. (The customary choice none is less fortunate:

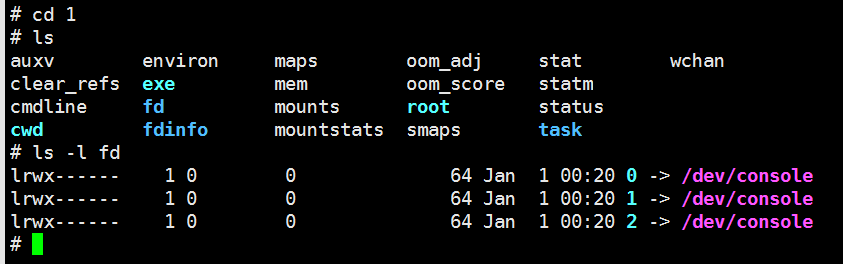
the error message `none busy' from umount can be con‐

fusing.)

当挂载了proc文件系统后在，proc目录下就会有相应的正在执行的文件。

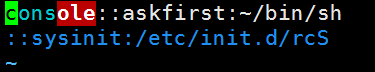


1 表示进程的PID号，可以到1中去, cd 1, fd就指向了标准输出，标准输入，标准错误。



如果不想手工的挂载就要加一个配置文件，在inittab中加就可以了。

在inittab中加上::sysinit:/etc/init.d/rcS，创建init.d目录，创建rcS的脚本，并给该脚本添加可执行权限。在该脚本中添加mount命令就可以了。

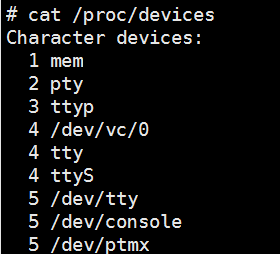


在rcS中可以用mount –t proc none /proc命令来挂载，也可以用mount –a 来挂载，-a 是挂载/etc/fstab文件中所列举的所有。所以直接在ftab中添加就可以了。

Mount –a 是读取/etc/fstab文件，根据fstab文件中的指示去挂载相应的文件系统。

启动后执行cat /proc mounts 查看挂载了那些文件系统。

Cat /proc devices 列出当前内核支持的那些设备，第1列是主设备号，当自己写的驱动(.ko)被insmod后就可以看到。



/dev中的文件对应设备和驱动，由于设备和驱动很多不可能手动去创建，所以采用udev机制(自动创建/dev/中的设备节点)，在busybox中有udev的简化版­——mdev

Here's a typical code snippet from the init script:

#挂载sysfs文件系统

[1] mount -t sysfs sysfs /sys

[2] echo /bin/mdev > /proc/sys/kernel/hotplug

[3] mdev -s

Of course, a more "full" setup would entail executing this before the previous

code snippet:

#挂载tmpfs文件系统

[4] mount -t tmpfs mdev /dev

[5] mkdir /dev/pts

[6] mount -t devpts devpts /dev/pts

字符设备

应用程序中使用的open, write , read，最终会调用内核中的xx\_open, xx\_write, xx\_read.那么应用层是怎么进入内核呢？open,write,read 会调用一条swi指令会引发异常进入内核。内核中有一个system call interface这个的作用是根据发生中断的原因，调用不同的处理函数。

./xxxx(进程) % 在后台运行

Top 命令相当于windows下的任务管理器