

嵌入式系统仿真实验第 11 讲

实验做到现在,我想我们需要开始做如何制作安装 linux 操作系统的实验了,但这部分不需要编程,也是最简单的部分,如何不会,网上多的是教程,所以今天这个操作系统的实验我将不花太多时间去讲解了。

1. 下载 linux 内核源代码,编译、调试

首先给大家一个参考网址,希望大家去这个网址对照看,我只挑我做了了一些实验步骤来验证实验。

参考网址: https://elinux.org/Virtual_Development_Board

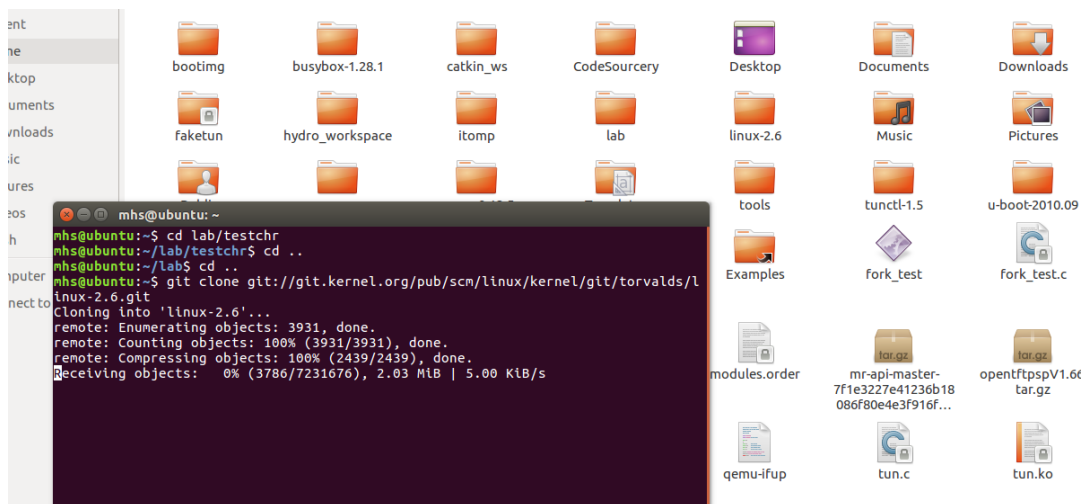


图 1 git clone linux 源代码

我没有选择下载,我来看看用 git 命令复制源代码下来看看。从图 1 可以看到下载速度比较慢,创建了个 linux-2.6 的文件夹,clone 下来的代码在这个目录里面。

等了半天,发觉 git clone 真的很慢,而且有时候几乎死机一般,小型源代码项目估计可以,这大型的项目我这网速不行啊,所以我这次换了一个方式,用 wget 命令去下个压缩包吧,这样应该会比较快。

wget <http://www.kernel.org/pub/linux/kernel/v3.0/linux-3.2.tar.bz2>

这次换了个 3.2 的版本内核文件去下载,见图 2,1 分 40 秒下载完成了,

```
mhs@ubuntu: ~  
HTTP request sent, awaiting response... 301 Moved Permanently  
Location: https://www.kernel.org/pub/linux/kernel/v3.0/linux-3.2.tar.bz2 [following]  
--2020-03-26 08:30:10-- https://www.kernel.org/pub/linux/kernel/v3.0/linux-3.2.tar.bz2  
Connecting to www.kernel.org (www.kernel.org)|147.75.46.191|:443... connected.  
HTTP request sent, awaiting response... 301 Moved Permanently  
Location: https://mirrors.edge.kernel.org/pub/linux/kernel/v3.0/linux-3.2.tar.bz2 [following]  
--2020-03-26 08:30:12-- https://mirrors.edge.kernel.org/pub/linux/kernel/v3.0/linux-3.2.tar.bz2  
Resolving mirrors.edge.kernel.org (mirrors.edge.kernel.org)... 147.75.95.133, 2604:1380:3000:1500::1  
Connecting to mirrors.edge.kernel.org (mirrors.edge.kernel.org)|147.75.95.133|:443... connected.  
HTTP request sent, awaiting response... 200 OK  
Length: 78147838 (75M) [application/x-bzip2]  
Saving to: 'linux-3.2.tar.bz2'  
  
linux-3.2.tar.bz2 100%[=====>] 74.53M 626KB/s in 1m 40s  
2020-03-26 08:31:52 (765 KB/s) - 'linux-3.2.tar.bz2' saved [78147838/78147838]  
  
mhs@ubuntu:~$
```

图 2 下载 v3.2 版本内核源代码包

下载下来大概 78.1M，先解压：

```
tar xjf linux-3.2.tar.bz2
```

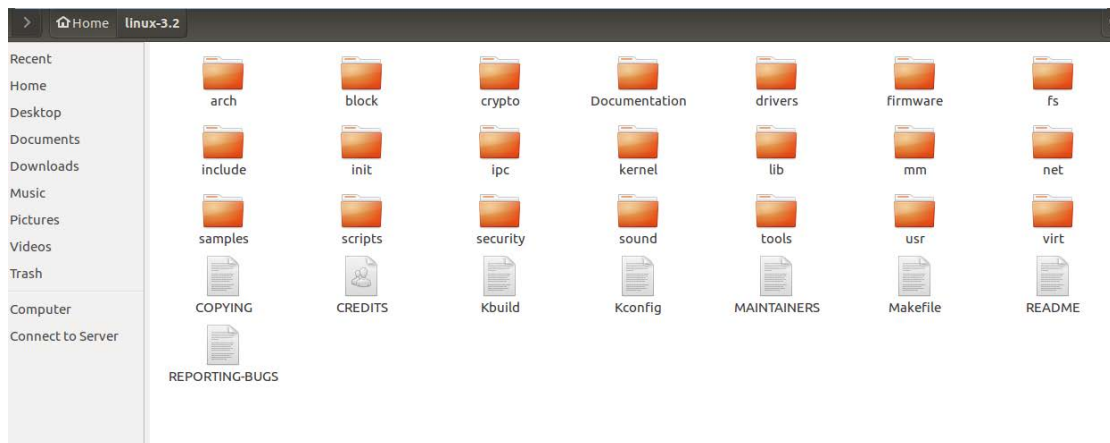


图 3 解压源代码

配置交叉编译器，以及编译配置文件：

```
mhs@ubuntu:~$ cd linux-3.2/  
mhs@ubuntu:~/linux-3.2$ make ARCH=arm  
CROSS_COMPILE=arm-none-linux-gnueabi- versatile_defconfig -s
```

配置 config 的时候这里有个错误，必须指定内核特征的交叉编译器选中 **ARM**

EABI, 所以我们执行这条语句:

make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- menuconfig

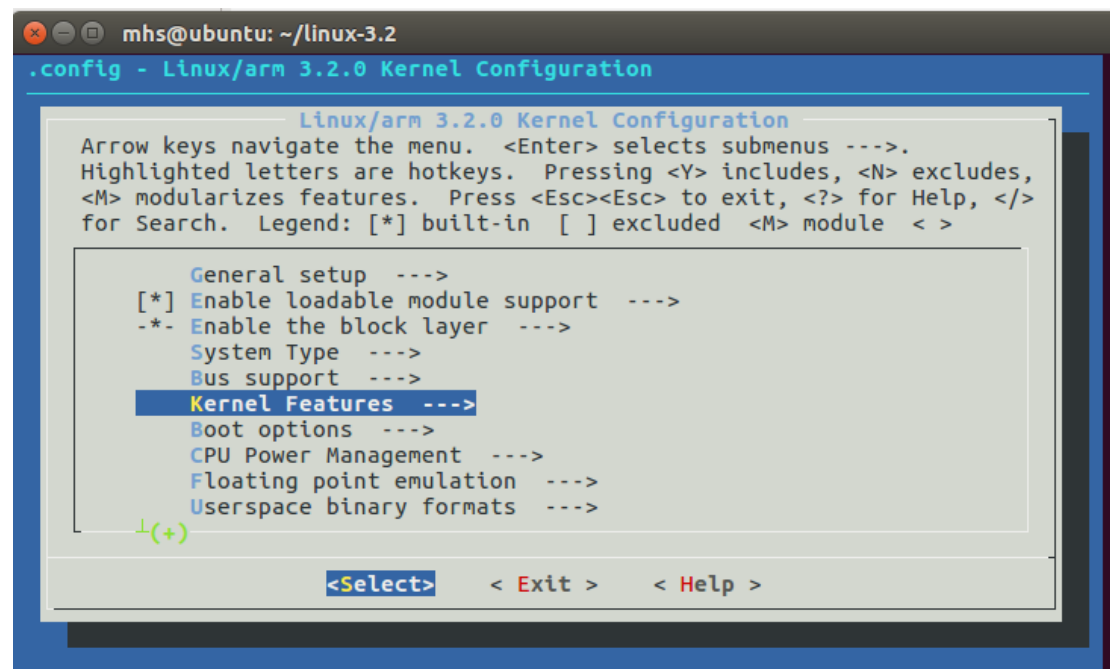


图 4 make menuconfig

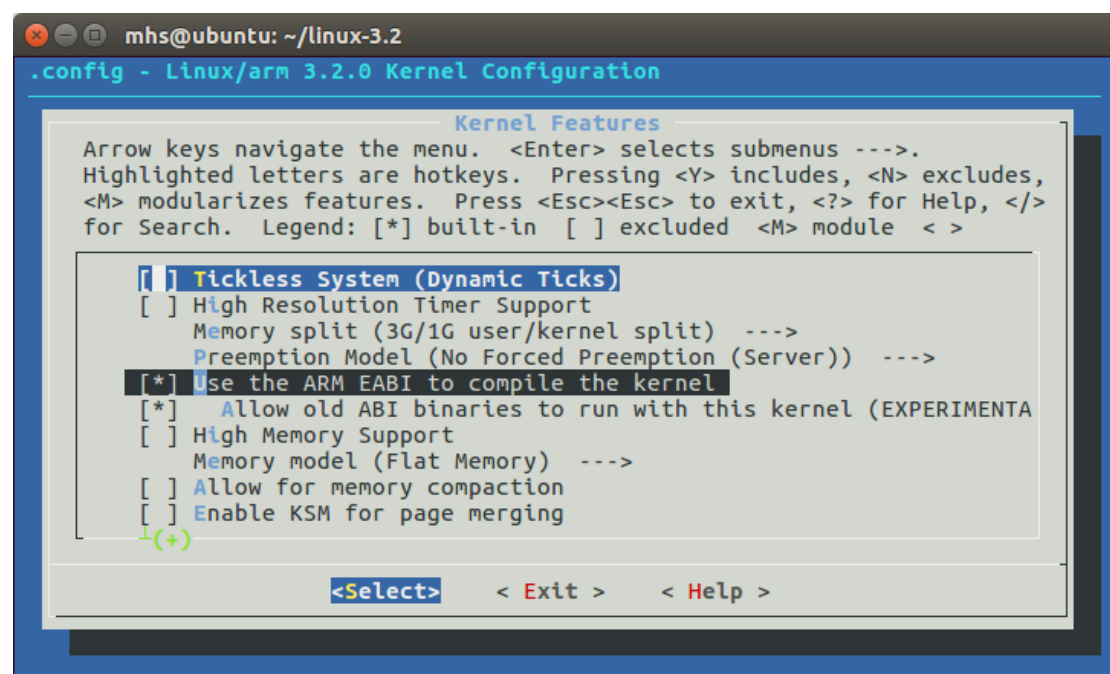
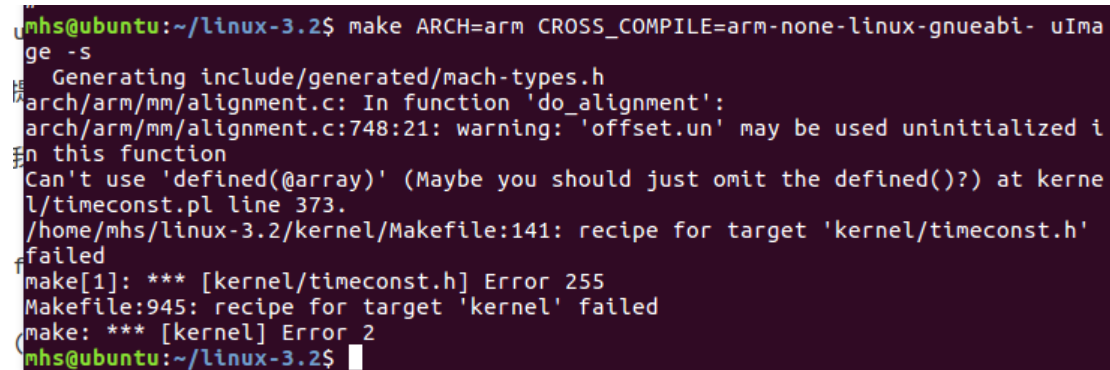


图 5 选中用 ARM EABI 交叉编译器编译内核

编译:

```
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- uImage -s
```

这个时候我发觉，出现了一个错误：



```
mhs@ubuntu:~/linux-3.2$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- uImage -s
Generating include/generated/mach-types.h
arch/arm/mm/alignment.c: In function 'do_alignment':
arch/arm/mm/alignment.c:748:21: warning: 'offset.un' may be used uninitialized in this function
Can't use 'defined(@array)' (Maybe you should just omit the defined()?) at kernel/timeconst.pl line 373.
/home/mhs/linux-3.2/kernel/Makefile:141: recipe for target 'kernel/timeconst.h' failed
make[1]: *** [kernel/timeconst.h] Error 255
Makefile:945: recipe for target 'kernel' failed
make: *** [kernel] Error 2
mhs@ubuntu:~/linux-3.2$
```

图 4 编译出现错误

有这么一句话：

Can't use 'defined(@array)' (Maybe you should just omit the defined()?) at kernel/timeconst.pl line 373.

其实，提示的错误信息已经明确告诉你了，你应该省略 `defined()`。

所以，我们打开 `kernel/timeconst.pl`，找到相应的地方 373 行：

```
@val = @{$scanned_values{$hz}};

if (!defined(@val)) {

    @val = compute_values($hz);

}

output($hz, @val);
```

将 `if (!defined(@val))` 改为 `if (!@val)`，再次编译就可以通过了。

这里我查了一下内核更新，发现其中有一项是 perl 版本升级到了 v5.22.1，然后查了 perl 官方文档，发现官网因为一个 bug，该版本将 `defined(@array)` 去掉了。可以直接使用数组判断非空，所以我就这么改了。

```

mhs@ubuntu:~/linux-3.2$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- uImage -s
drivers/mmc/card/block.c: In function 'mmc_blk_issue_rq':
drivers/mmc/card/block.c:794:15: warning: 'from' may be used uninitialized in this function
drivers/mmc/card/block.c:794:21: warning: 'nr' may be used uninitialized in this function
drivers/mmc/card/block.c:794:25: warning: 'arg' may be used uninitialized in this function
drivers/mtd/chips/cfi_cmdset_0001.c: In function 'cfi_inteltext_write_words':
include/linux/mtd/map.h:330:11: warning: 'r$x$0' may be used uninitialized in this function
Kernel: arch/arm/boot/Image is ready
Kernel: arch/arm/boot/zImage is ready
Image Name:      Linux-3.2.0
Created:         Thu Mar 26 09:18:45 2020
Image Type:      ARM Linux Kernel Image (uncompressed)
Data Size:       1917320 Bytes = 1872.38 kB = 1.83 MB
Load Address:    00008000
Entry Point:     00008000
Image arch/arm/boot/uImage is ready
mhs@ubuntu:~/linux-3.2$

```

图 5 编译完成

编译出现了几个 mmc 下四个参数可能未初始化的警告，可以不用理它。编译成功生成了 uImage 文件，是个未压缩的映像文件，大概 1.83MB。

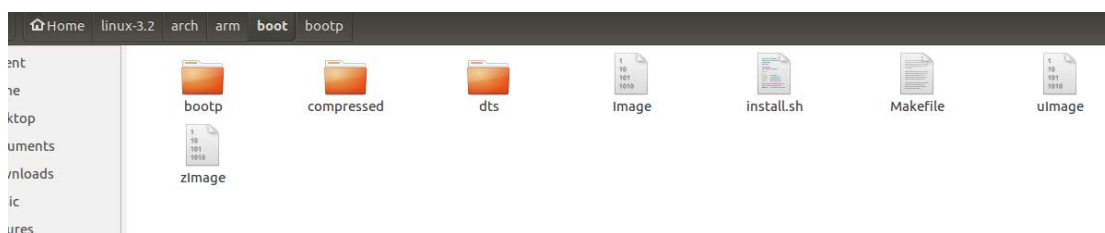


图 6 编译成功生成 uImage

从图 6 我们看到其实在 "arch/arm/boot/" 目录里还生成了 image 和 zImage 文件。

接下来我们返回到 home 目录，编写一个测试小程序：

init.c:

```

#include <stdio.h>

void main() {
    printf("Hello World!\n");
    while(1);
}

```

编译一下：

```
arm-none-linux-gnueabi-gcc -static init.c -o init
```

注意加入-static 参数，然后我们想把这个 init 放到 ramfs 里面去，在初始化 linux 操作系统的时候执行，接下来我们做一个 ramfs 文件：

```
echo init|cpio -o --format=newc > initramfs
```

```
mhs@ubuntu:~/linux-3.2$ cd ..
mhs@ubuntu:~$ echo init|cpio -o --format=newc > initramfs
1267 blocks
mhs@ubuntu:~$ cpio -t < initramfs
init
1267 blocks
mhs@ubuntu:~$
```

图 7 使用 cpio 建立一个 ramfs 文件

我们看到 home 目录生成了一个 initramfs 文件，里面包含 init 文件共 1267 个 blocks。

接下来我们使用 qemu 仿真运行这个 linux 内核，并初始化 ramfs，执行初始化程序 init：

```
qemu-system-arm -M versatilepb -kernel linux-3.2/arch/arm/boot/zImage
-initrd initramfs -serial stdio -append "console=ttyAMA0"
```

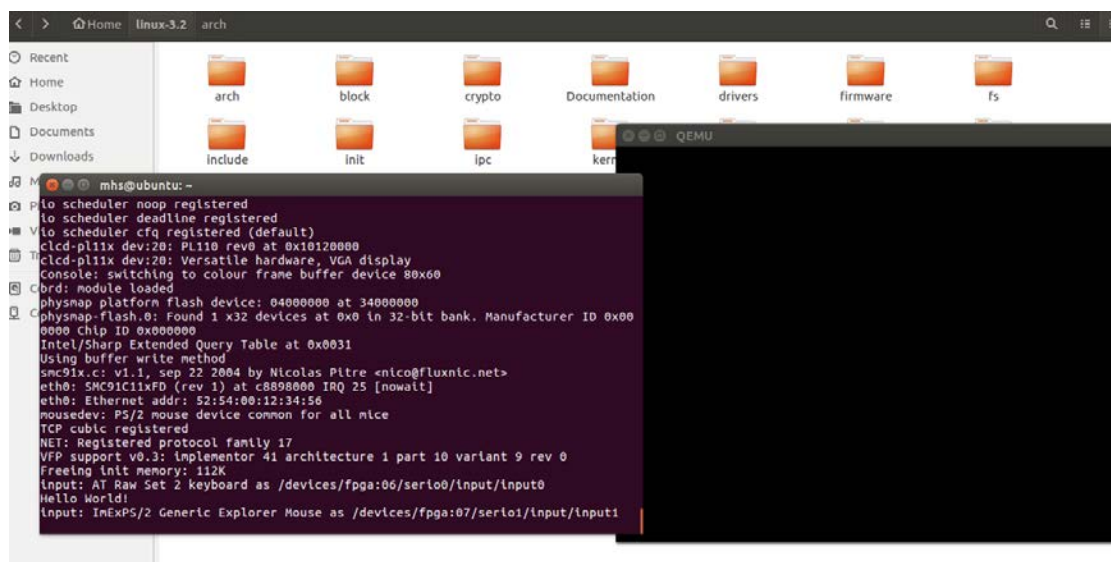


图 8 仿真运行内核文件及 ramfs 初始化文件

从图 8 中，我们可以看到，qemu 就好比是一个 bootloader 不但仿真运行内

核，还引导内核挂接了 `ramfs`，执行了 `init` 这个程序。

如果我们要制作一个完整的系统软件，除了上一次课我们制作的 `U-Boot` 作为 `bootloader`，本节的 `linux` 内核外，`linux` 运行起来实际上还需要我们制作一个文件系统。

2 Busybox

`Busybox` 可用于需要紧凑文件系统的嵌入式 `Linux` 设计解决方案，它可以把许多系统程序编译并链接到一个二进制文件中，比如我们用它来构建嵌入式系统的根文件系统。通常，一个有效的 `Linux` 根文件系统包含一个小的目录树（`/ bin`，`/ sbin`，`/ usr / bin` 等）。生成单个可执行二进制文件在 `/ bin / busybox` 中，以及许多与 `Busybox` 二进制文件的符号链接（`/ bin / ls`，`/ bin / sh`，`/ sbin / ifconfig` 等），如果使用典型配置可以将其减小到 `2MB`（如果压缩则为 `1MB`）。

`Busybox` 可以到 <https://busybox.net/downloads/> 去下载，这次我选的 `busybox-1.28.1.tar.bz2` 这个包，解压，`tar xjf busybox-1.28.1.tar.bz2`，然后进入到解压的源代码目录，执行编译前的配置：

先初始化配置文件：

```
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- defconfig
```

然后配置交叉编译选项：

```
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- menuconfig
```

选中将 `Busybox` 编译为静态可执行文件的选项，这样我们就不必在根文件系统中复制动态库。该设置可以在“`Busybox Settings --> Build Options`”中找到，见图 9。

同时，注意，我们需要把 `Coreutils--->sync` 选项去掉，以及 `Linux System Utilities--->nsenter`，也要去掉该选项，确保下一步的编译安装没有任何错误。

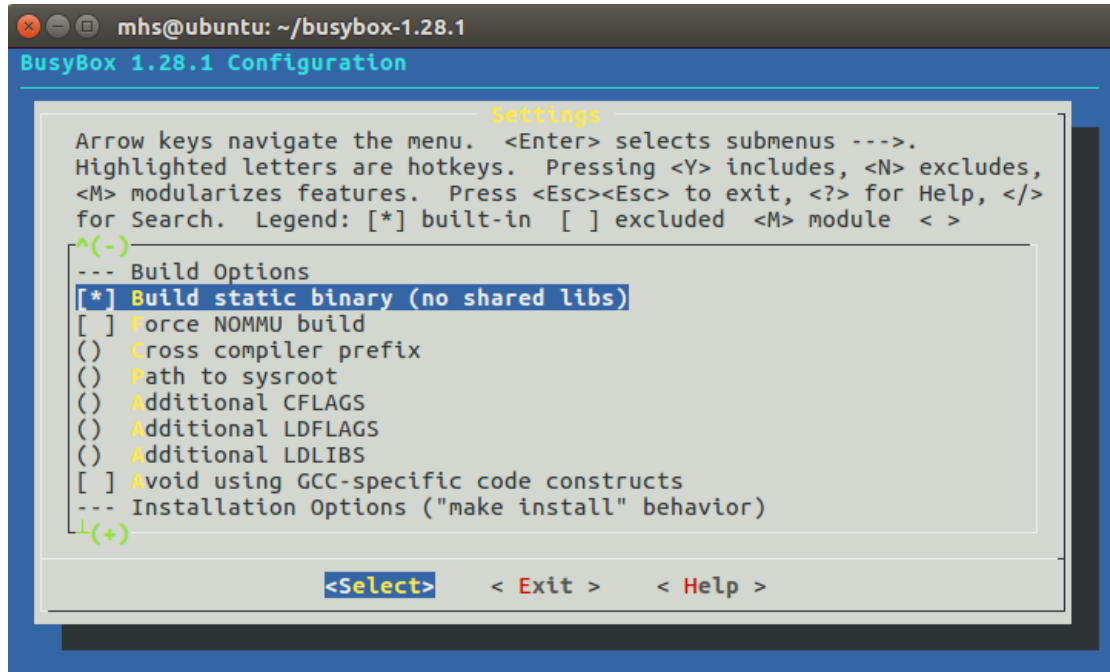


图 9 配置 busybox 编译选项

保存退出后执行下面的命令,将构建 Busybox 并创建一个名为_install 的目录,其中包含根文件系统树:

```
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- install
```

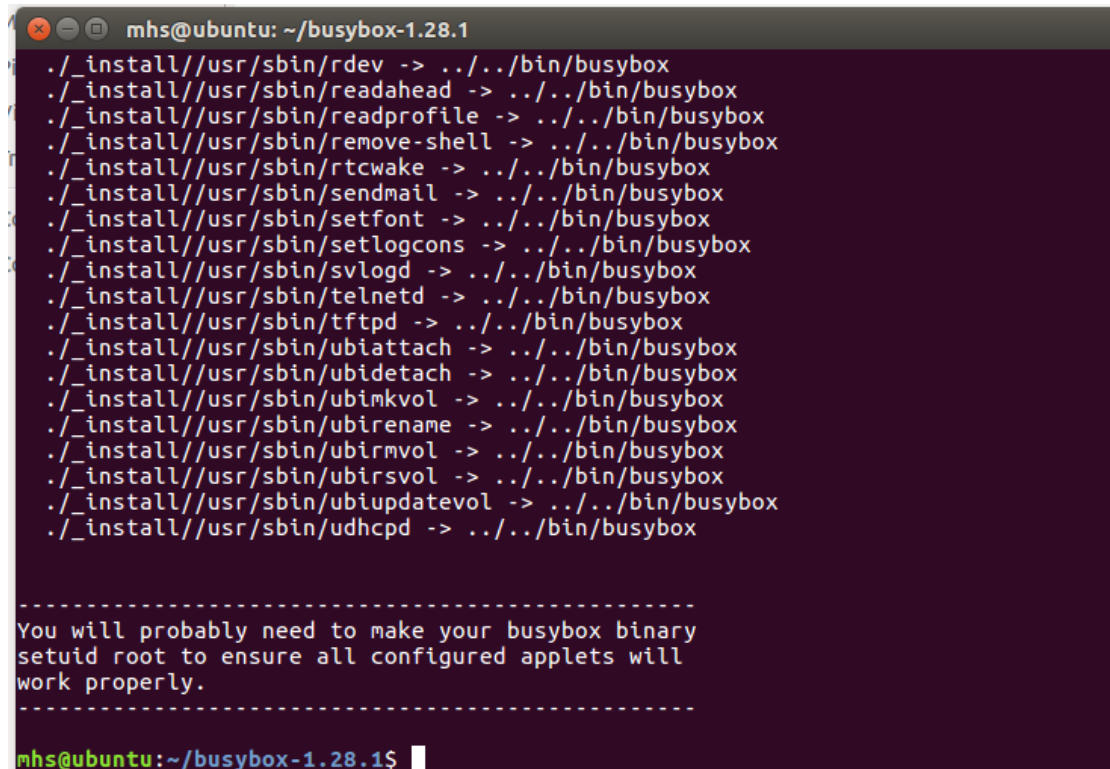


图 10 成功编译 busybox 并生成了根文件系统的安装目录_install

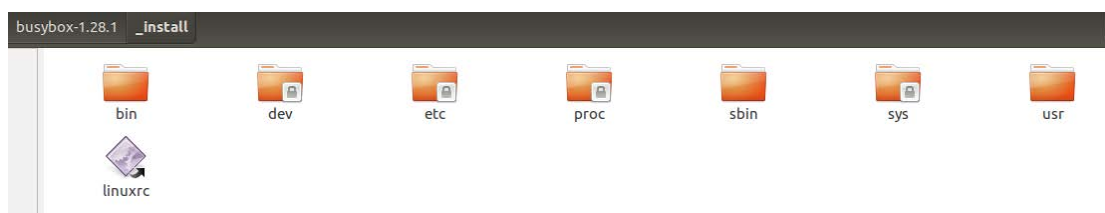


图 11 busybox 生成的安装目录

接下来我们还是使用 `cpio` 工具生成我们需要的根文件系统, 并使用 `gzip` 把它压缩:

```
$ cd _install
$ find . | cpio -o --format=newc > ../rootfs.img
$ cd ..
$ gzip -c rootfs.img > rootfs.img.gz
```

我们回到 `home` 目录, 开始仿真执行上一步的内核挂接这个文件系统:

```
qemu-system-arm -M versatilepb -m 128M -kernel
linux-3.2/arch/arm/boot/zImage -initrd busybox-1.28.1/rootfs.img.gz
-append "root=/dev/ram rdinit=/bin/sh"
```

执行之后, 我们看到这次是在 `qemu` 的界面里做终端输出, `linux` 内核正确启动, 并且成功挂接了根文件系统, 光标停留在 `#` 后面, 见图 13, 接着我们就可以操作一下 `linux` 的一些常用命令看看, 有没有把它们裁剪下来。



图 12 测试 linux 命令

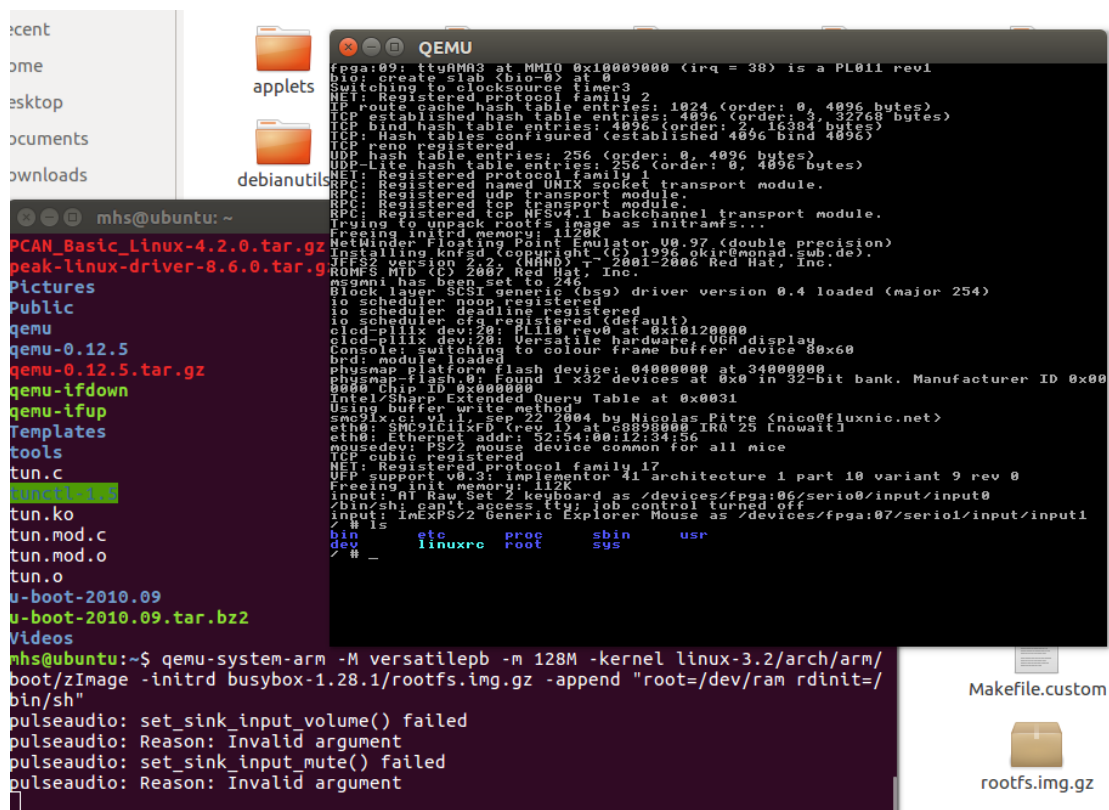


图 13 仿真内核挂载根文件系统

从这里我们可以看到无论是 U-Boot 挂载个 ramfs 还是让内核挂载个 ramfs 我们都可以仿真运行起来，毫无问题。

接下来我们要看看，如果是把 U-Boot 和 kernel 以及根文件系统一起烧写到一个文件里，是否可以用 qemu-system-arm 一起去仿真运行呢，如果这样可行的话，我们就免得像很多方案里去搞 tftp 服务器，或者配 nfs 去玩仿真了，因为这些对初学者来说可能会更复杂，更难以上手，特别是 vmware 虚拟机下的问题很麻烦，我记得我好几年前为解决 tun0 的问题都差不多把 ubuntu 和 vmware 的 bug 翻了个透才解决的，现在手头就一个很古老的 macbook air，硬盘还都是满的，幸运的是有个 vmware workstation，我连 Ubuntu 虚拟机都是装在 U 盘上的，如果又要配 nfs、tftp 之类实在不想再回忆再弄。既然 qemu 的 ramfs 没有问题，我们觉得还是用 /dev/ram 的方式去仿真吧。

3 U-Boot+zImage+rootfs 一起仿真

前面我们都已经有了这三个文件了，现在我们是不是很简单就可把它们揉在

一个二进制文件里去一起仿真运行呢，前面我们用过 `cpio`，现在还有这个行么，想想估计不行，为什么因为 `bootloader` 启动 `linux` 操作系统是需要带启动参数才能运行起来的，这个我们在理论课里面讲过。

典型存储空间分配结构

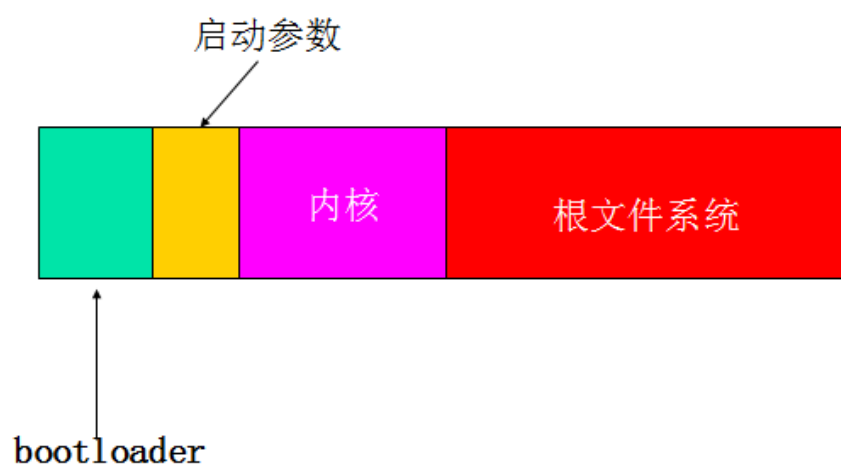


图 14 存储空间分配结构图

图 14 是我们理论课里面的一张图，对于这个启动参数来说，需要由 `bootloader` 传递给内核。具体用 `U-Boot` 的时候可以使用 `append` 参数给 `kernel`，比如我们前面用过的命令：`-append "console=ttyAMA0"`

最重要的是我们需要传给 `kernel`，指定挂接的 `rootfs` 在哪里，初始化运行哪个地方的程序等等，这要是前面把三个文件打包到一起，怎么传递呢？

网上找到一张图，我直接拿过来参考一下，图 15 为我们如果要用 `qemu` 去仿真运行 `U-Boot`、内核以及根文件系统时，这三个文件在内存中的分布情况。

从图 15 中，我们很早就知道 `qemu` 仿真 `arm926ejs` 这个 CPU 核的板子，它的起始地址就是 `0x00010000`，也就是如果我们第一个运行的是 `U-Boot`，那么 `U-Boot` 就是从 `0x00010000` 开始，我们知道 `u-boot.bin` 文件大概 87.6K 大小，我们考虑多一点，把内核从 2M 开始存放，这样的话，内核偏移址 `0x200000`，根文件系统从 4M 的地方开始存放，加 `qemu` 加载开始就有的偏移址 `0x10000`，

那就是 0x410000 开始就是根文件的起始内存地址。

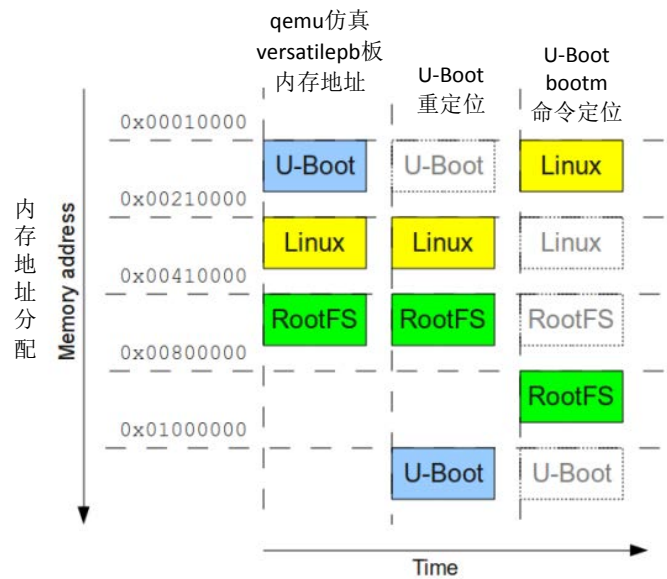


图 15 三个过程各文件在内存中存放的地址

第一个阶段由 `qemu-system-arm` 把文件加载到内存，第二个阶段，开始运行 U-Boot，U-Bootd 的运行机制我们前面介绍过，它第一个阶段初始化环境后转入 C 语言，把它自己本身拷贝到高端内存 0x01000000 保护起来，不让一般用户程序访问破坏，到第二个阶段(stage2)也就是我们现在说的第三个阶段前，U-Boot 需要将系统运行权完全交给 linux 操作系统内核，一般将操作系统的代码解压拷贝到系统的最前面，这里 qemu 最开始的地址是 0x10000（注意，如果是实物，这个地址将是 0x000000），在把启动参数传给 linux 内核之前，还会将根文件系统映射到比如 0x00800000 处，然后把 PC 指针转向最低地址处，开始运行操作系统，把 CPU 的运行权完全交给操作系统程序。

明白这个原理，我们将前面生成的 `u-boot.bin` `zImage` `rootfs.img.gz` 三个文件拷贝到一个目录比如 `ex16`，开始打包：

```
mkimage -A arm -C none -O linux -T kernel -d zImage -a 0x00010000 -e 0x00010000 zImage.uimg
mkimage -A arm -C none -O linux -T ramdisk -d rootfs.img.gz -a 0x00800000 -e 0x00800000 rootfs.uimg
dd if=/dev/zero of=flash.bin bs=1 count=6M
dd if=u-boot.bin of=flash.bin conv=notrunc bs=1
dd if=zImage.uimg of=flash.bin conv=notrunc bs=1 seek=2M
```

```
dd if=rootfs.uimg of=flash.bin conv=notrunc bs=1 seek=4M
```

mkimage 两个命令我们前面使用过了，将 zImage 和 rootfs.img.gz 这两个文件准备成带符号的映像文件，而且指定了它们预计最后要加载到内存的目的起始地址是 0x00010000 和 0x00800000。四个 dd 命令创建了一个 6M 大小空间的 flash.bin 文件，起始是 u-boot.bin，然后从 2M 开始的地方放 zImage.uimg，从 4M 开始的地方放 rootfs.uimg。

用 qemu-system-arm 去启动仿真运行看看吧：

```
qemu-system-arm -M versatilepb -serial stdio -kernel flash.bin
```

有点不幸的是，我们无论如何都在 U-Boot 里面去 setenv 配置启动参数，都无法让内核挂接成功根文件系统，这是为什么呢？

使用 iminfo 0x210000 以及 iminfo 0x410000 都可以看到内核和根文件系统都是正确的分配在内存中，但系统无法自动启动，停留在 versatilepb:提示符下，无论如何 setenv 修改 bootargs 参数，系统总是不起作用，load 内核和根文件系统显示没有错误，但操作系统总无法从 ram 里挂接根文件系统，总是报错：

```
vfs: unable to mount root fs via NFS, trying floppy
```

这是为什么呢，这是因为 U-Boot 和 qemu 之间在 versatilepb 这块板的仿真方面有个 BUG，BUG 出现在 U-Boot 源代码目录里面/common/image.c 里面，在 int boot_get_ramdisk 这个函数里面有这么一句话：

```
#if defined(CONFIG_B2) || defined(CONFIG_EVB4510) ||  
defined(CONFIG_ARMADILLO)
```

看这里没有 versatilepb，所以我们需要修改这个地方为：

```
#if defined(CONFIG_B2) || defined(CONFIG_EVB4510) ||  
defined(CONFIG_ARMADILLO) || defined(CONFIG_VERSATILE)
```

加入 CONFIG_VERSATILE。

还有一个地方在/include/configs/versatile.h 这个文件里面，它是这么定义的 boot args 参数的：

```
#define CONFIG_BOOTARGS "root=/dev/nfs mem=128M ip=dhcp "\
```

不知道，为何我们在 U-Boot 里面用 `setenv bootargs` 命令修改，用 `printenv` 显示修改成功，但无论如何这个 `boot args` 不胜成效，多半的原因是有前面没有 `define` 的缘故，这里为了保险，我们让系统能自动启动挂接根文件，而不是停留在 U-Boot 命令行需要手动修改 `boot args` 参数，我们直接在这里把缺省的启动参数修改成：

修改完成后，我们需要把 U-Boot 按前面的教程，配置、编译，生成 u-boot.bin 文件，然后把上面打包过程重做一边，然后运行测试：

如果在这句话 Hit any key to stop autoboot: 0 耗完之前没有按键，那么它就会直接启动整个系统，如果按键就会停留在 U-Boot 的命令行。

```

mhs@ubuntu:~/lab/ex16
VersatilePB # printenv
bootargs=root=/dev/ram ram=128M rdinit=sbin/init
bootcmd=bootm 0x210000 0x410000
bootdelay=2
baudrate=38400
bootfile="/tftpboot/uImage"
stdin=serial
stdout=serial
stderr=serial
verify=no
ethact=SMC91111-0

Environment size: 205/8188 bytes
VersatilePB # bootm 0x210000 0x410000
## Booting kernel from Legacy Image at 00210000 ...
Image Name:
Image Type: ARM Linux Kernel Image (uncompressed)
Data Size: 1939176 Bytes = 1.8 MiB
Load Address: 00010000
Entry Point: 00010000
## Loading init Ramdisk from Legacy Image at 00410000 ...
Image Name:
Image Type: ARM Linux RAMDisk Image (uncompressed)
Data Size: 1150276 Bytes = 1.1 MiB
Load Address: 03000000
Entry Point: 03000000
Loading Kernel Image ... OK

OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.

```

至此,我们的嵌入式系统仿真实验算是完全到了系统级,再也不是 bare metal 了,后面的章节我们会介绍 linux 下的设备驱动程序编写,这个设备驱动程序编

写，在 ARM 环境和 PC 环境下没有什么本质区别了，不同在于编译器不同而已。有了打包运行的方法，我们后面测试 ARM 环境的 linux 设备驱动也不会有什么困难。

References

- [Using U-Boot and Flash emulation in QEMU](#)
- [U-boot for ARM on QEMU](#)
- [Debugging Linux systems using GDB and QEMU](#)
- [Virtual Development Board](#)
- [Busybox for ARM on QEMU](#)

注：本实验教程为武汉科技大学机器人与智能系统研究院闵华松老师的网络课程教学文档，请遵守 MIT 协议，可以复制，不做商业用途，转载请注明出处。