Linux 内核移植

姓名： 周余 （nack）
实验室： 南大蔡冠深软件研发中心
研究方向： 嵌入式系统、图像处理
EMAIL: nackzhou@sw-linux.com

报告主要内容

1. **Linux** 内核的基本概念

2. **Linux** 内核移植的主要工作

3. **Linux** 内核源代码介绍

4. **Linux** 内核移植的实例

5. **Linux** 内核的裁剪

6. 文件系统及 **init** 以后的启动

内核的一些基本概念

进程的基本概念

linux 中每一个进程由一个 **task_struct** 结构描述，定义于 **\include\linux\sched.h.** 也就是 **PCB.** 其中包含了系统用来管理进程的所有信息：进程状态，调度信息（优先级），各种标识， **IPC** 有关信息，时间和定时信息，文件系统西信息， **VM** 信息，页面信息，环境 **context** 等 **.**

一个进程包含两种结构：用户态下的数据和堆栈；内核态的堆栈和 task_struct. task_struct 和内核栈存放在一个 8K 大小的内存块中，其低地址存放 task_struct，然后由高到低是内核态的堆栈。当一个进程在 CPU 上执行的时候，通过将当前堆栈指针低 13 位清 0 来得到指向当前进程 task_struct 的指针（因为内存块大小是 8K，低 13 位清 0 就是该内存块的最低地址）。由 current 宏来实现，指向当前 CPU 上运行的进程的 task_struct.

内核的一些基本概念

进程的基本概念

进程的组织形式： linux 中进程分为用户进程和内核线程，都有自己的 task_struct.linux 用以下几种方式来管理进程：

a. 进程链表：是动态的，更加灵活和方便。 linux 为每一种状态的进程都维护了一个双向链表，这样可以更方便管理。例如，在要选择一个就绪的进程的时候就直接在 TASK_RUNNING 链表中选择一个，而不用去遍历整个进程链表。链表每一个结点就是一个 task_struct 。 task_struct 中的 next_task 和 prev_task 指针来实现链表。

b. TASK_RUNNING 状态链表：当内核要寻找一个新的进程在 CUP 上运行的时候，必须寻找能立即投入运行的进程，也就是 TASK_RUNNING 状态的进程。

c. 等待队列：等待队列同 TASK_RUNNING 队列不同，它有自己特殊的结构：

```
sreuct wait_queue{
        struct task_struct  *task;
        struct wait_queue   *next;
        };
```

等待队列表示一组睡眠的进程，当某个条件为真的时候由内核唤醒它们。

d. task 数组空闲表项的链表：当一个进程创建或撤消的时候， task 数组必须被更新。线性的搜索太低效，内核就维护了一个非循环双向链表，包含 task 数组中的空闲项。当创建一个进程的时候直接从这个链表取走第一个表项；当删除一个进程的时候将空闲表项加入链表的第一项。

e. HASH 表：在 linux 中的 HASH 表叫做 pidhash ，因为是由进程的 PID 映射的。它由 PIDHASH_SZ 个元素组成，表项是指向 task_struc 的指针。对于 HASH 的冲突，将各由冲突的表项连成双向链表。

其中 a, e 是针对系统中所有进程的，而 b, c, d 是将具有同一状态的进程组织起来。

内核的一些基本概念

Linux 的进程调度

在 linux 中进程的优先级是动态的，调度程序跟踪并适时调整进程的优先级。在 linux 中进程是抢占式的，而内核是非抢占的。

* 所谓进程是抢占的也就是说如果有一个进程进入 TASK_RUNNING 状态，内核将检查它的优先级是否大于 current 的优先级，若大于，则 current 被中断，并用调度程序选择一个进程运行（注意！不一定是刚刚那个 TASK_RUNNING 进程，但通常是这样）。当然，在一个进程时间片到期的时候也可以被抢占。
* 所谓内核非抢占是说内核态下的进程不能被随意中断，以保证内核态进程数据的完整和有效。也避免了一些同步的问题。

内核的一些基本概念

Linux 的进程调度

linux 所选择的进程调度算法将 CPU 时间划分为时期，在一个时期内，每一个进程有一个指定的时间片。通常每个进程的时间片不相等。当一个进程的时间片用完时，将被调度。在一个时期内进程也可以多次被调度，只要它的时间片还没有用完。当所有进程都用完他们的时间片的时候，这个时期结束。调度算法重新计算每个进程的时间片，开始一个新的时期。

那 linux 在什么情况下才进行调度呢？

1. 进程状态的改变。例如 RUNING->STOPPED 。
2. 当前进程时间片用完。
3. 设备驱动程序。
4. 进程从中断，异常，系统调用返回的时候。

对于其中的 1 ，进程要调用 exit(),wait() 等函数，这些函数自动调用调度程序；对于 3 ，当驱动程序执行长而重复的任务时候直接调用调度程序。在每次反复循环中都由驱动程序检查 need_resched 的值，如果有必要就调用调度程序。。而对于 2 和 4 ，由于都是从中断或调用返回，最终都要调用 ret_from_sys_call() ，有这个函数进行判断，有必要就进行调度

内核的一些基本概念

Linux 的进程的切换

进程的切换主要包括地址空间的转换和上下文的切换。先说上下文切换

在操作系统课程中我们学过每一个进程的 **TSS** 段都保存了它的硬件上下文（**hardwarecontext**），当由中断触发进行进程切换的时候，**CUP** 会自动将任务门中的段选择符装入 **TR** 寄存器，使 **TR** 指向新的 **TSS**，完成任务切换。但由于硬件自动装入和切换会带来很多不必要的开销，所以 **linux** 选择了用软件来实现进程切换。**TSS** 在 **linux** 中没有实际作用，只是在内核初始化的时候设置一个 **TSS**，以后全部用这个 **TSS**，只是通过更换其中的部分内容来切换进程。**linux** 进程切换主要是通过切换每一个任务在 **TSS** 中的内核栈指针 **SS0** 和 **ESP0** 来实现的。具体的切换依赖于 **switch_to**（）这个宏

下面说地址空间的切换：主要由 \include\asm-i386\Mmu_context.h

static inline void switch_mm(struct mm_struct *prev, struct mm_struct *next, struct task_struct *tsk, unsigned cpu)

内核的一些基本概念

Linux 的内存管理，虚拟内存可以提供以下的功能：

广阔的地址空间：系统的虚拟内存可以比系统的实际内存大很多倍。

进程的保护：系统中的每一个进程都有自己的虚拟地址空间。这些虚拟地址空间是完全分开的，这样一个进程的运行不会影响其他进程。并且，硬件上的虚拟内存机制是被保护的，内存不能被写入，这样可以防止迷失的应用程序覆盖代码的数据。

内存映射：内存映射用来把文件映射到进程的地址空间。在内存映射中，文件的内容直接连接到进程的虚拟地址空间。

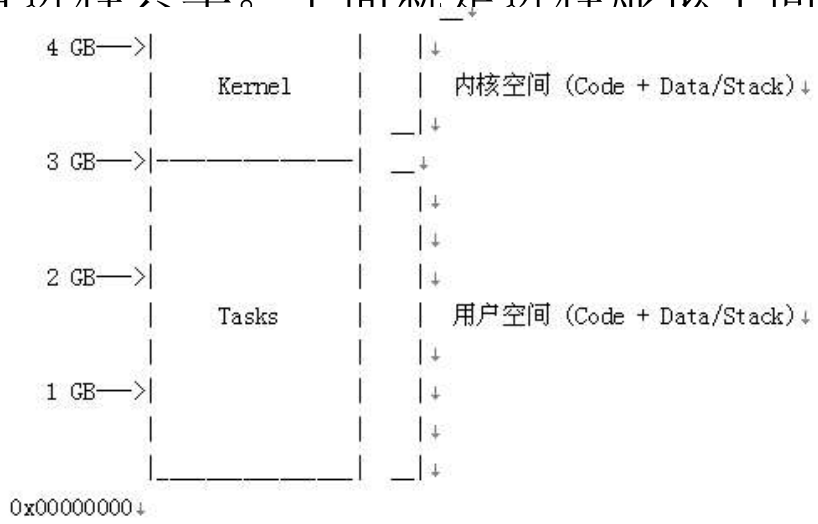
公平的物理内存分配：内存管理系统允许系统中每一个运行的进程都可以公平地得到系统的物理内存。

共享虚拟内存：虽然虚拟内存允许进程拥有自己单独的虚拟地址空间，但有时可能会希望进程共享内存。

内核的一些基本概念

Linux 的内存管理，内核空间和用户空间

linux 内核采用了分段机制，使得虚拟地址和线性地址总是一致，因此， linux 的虚拟地址空间为 0 — 4G 字节。 Linux 将这 4G 字节的空间分为两部分。将最高的 1G 字节（从 0XC0000000 到 0XFFFFFFFF ），供内核使用，称为“内核空间”。而将较低的 3G 字节（从 0X00000000 到 0XBFFFFFFF ），供各个进程使用，称为“用户空间”。因为每个进程可以通过系统调用进入内核，因此， linux 内核由系统的所有进程共享。下面就是进程虚拟空间示意图。



Linux 内核移植的主要工作

内核移植一般分为四部分工作：

- CPU Core 支持
- CPU 支持及 SOC 片上系统支持
  外围设备驱动程序开发

内核移植的开发步骤：

1. 阅读硬件文档

- CPU 体系结构及汇编
  地址空间分布
  寄存器列表及详细功能
  硬件原理图

Linux 内核移植的主要工作

2、建立交叉编译环境

有二种方法：

从网上获得编译好的二进制代码

自己生成（ToolChain-HOWTO）

- binutils
- gcc
- glibc/newlib/uclibc

```
#!/bin/sh
Export PATH=/usr/local/arm/2.95.3/bin/:$PATH
```

Linux 内核移植的主要工作

3、编写 Bootloader

Bootloader 的基本功能主要下面几个：

硬件初始化（时钟、Flash、SDRAM、串口、片选等）

实现一种通信协议 (xmoden, zmoden, or a small protocol that user defined)

下载内核、文件系统及应用程序到 Flash 中

内核及文件系统搬运

内核启动

Bootloader 程序难点：

硬件初始化

擦写 Flash

Linux 内核移植的主要工作

4 、内核移植和裁剪

在最开始时，将内核最基本的功能保留，其余的全部去掉，用来调试内核

5 、建立文件系统

最开始采用 ramdisk 来调试，当 MTD ， IDE 驱动完成后再使用其他块设备

6 、驱动开发 （下一节课详细介绍）

Linux 内核源代码介绍

Linux 内核源代码是公开的，标准内核可以在以下网站上得到：

ftp://ftp.kernel.org/pub/

http://www.kernel.org/pub/

针对某一类型 CPU ，同样有相应的 patch

ARM ： ftp://ftp.arm.linux.org.uk/

…

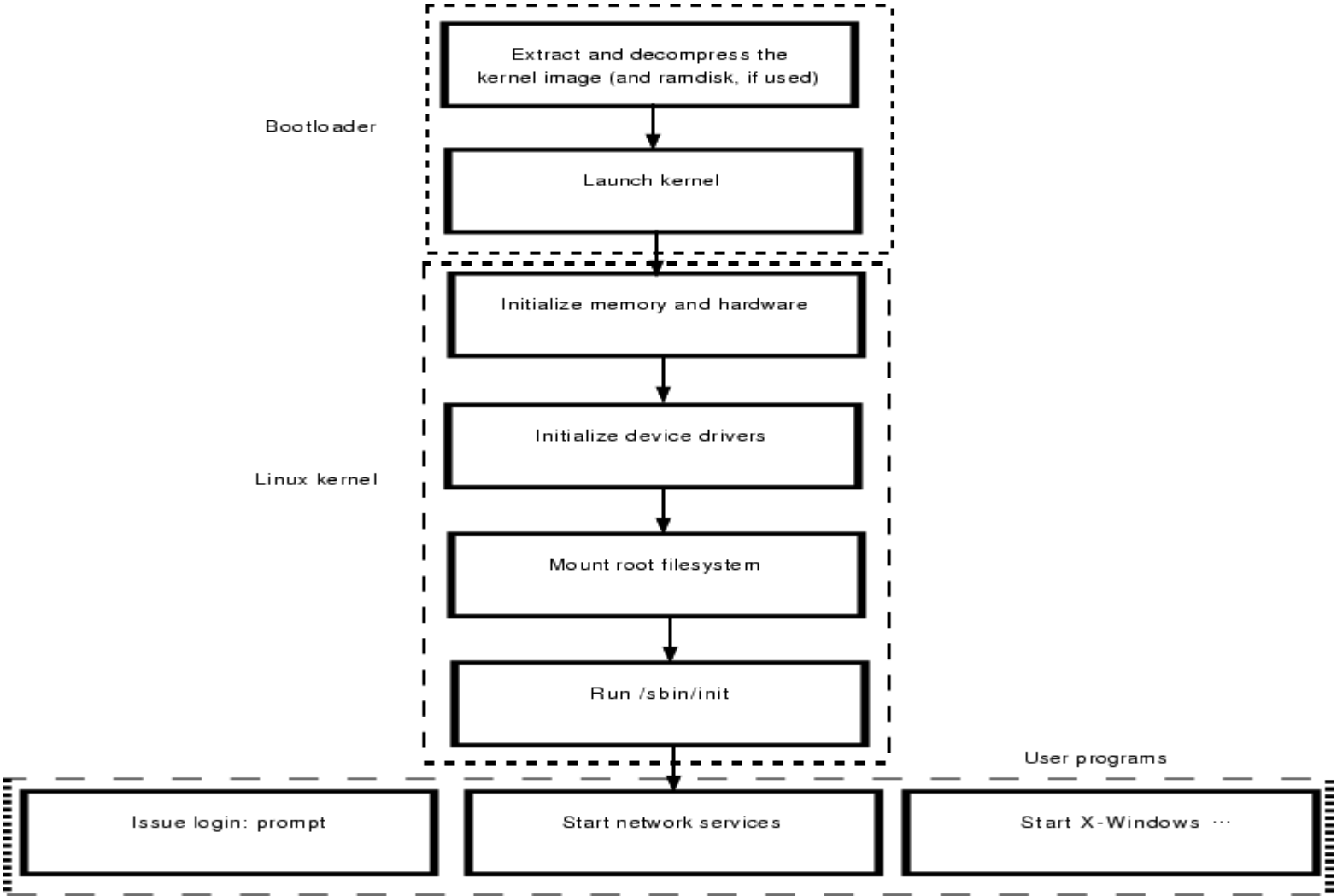
Linux 内核源代码介绍

Linux 内核源代码包含以下子目录：

- arch- 这个目录包含与体系结构相关的代码。对每个支持的体系结构（MIPS， ARM， 386，等等），在"arch" 下有一个相应的子目录。每个体系结构的子目录下有四个主要的子目录：
  - kernel，包含与体系结构相关的内核代码
  - mm，包含体系结构相关的内存管理代码
  - lib，包含与体系结构相关的库代码 (vsprintf 等等)
  - mach-（目标平台目录），包含基于此体系结构平台相关的代码
- documentation- 包含内核的文档
- drivers- 包含设备驱动代码。每类设备有相应的子目录，如 char, block, net，等等
- fs- 包含文件系统的代码。对每个支持的文件系统 (ext2, proc,JFFS2 等等) 有相应的子目录
- include- 包含内核的头文件，对每一种体系结构，分别有相应的子目录 asm- *
- init- 包含内核的初始化代码
- kernel- 包含内核代码
- lib- 包含内核的库代码
- mm- 包含内核管理代码

Linux 内核移植的实例

系统启动顺序



Linux 内核移植的实例

Bootloader

Bootloader 一般分为两个 stage：

第一个 stage 根据开发板的硬件资料做一些初始化的工作（汇编）：

初始化 PLL,

初始化存储空间， SDRAM, 地址空间分配

初始化串口， LED 等基本调试硬件

设置系统运行模式，设置各种模式的堆栈

第二个 stage 实现 bootloader 通用的一些功能（C）

用户界面（命令行）

串口驱动，网口驱动， usb 驱动， flash 驱动

下载协议 xmoden,tftp

下载功能（内核， ramdisk 等），烧写 flash

Linux 内核移植的实例

内核启动：

Boot code:

arch/arm/boot/compressed/head.S

arch/arm/boot/compressed/head-xxx.S

arch/arm/boot/compressed/setup-xxx.S

Arch/arm/kernel/head-armv.S

Init/main.c

Arch/arm/mach-xxx

Include/asm-arm/arch-xxx

Linux 内核移植的实例

汇编代码 linux/arch/arm/kernel/head-armv.S 是内核代码的入口，连接时它位于内核代码最前面，内核从这儿执行，查找 processor 和 architecture 类型、建立内核的初始化堆栈、建立临时页表，清除 BSS 段，然后 TLB 和 caches 被刷新，最后进入 start_kernel 。

```
mov r0, #F_BIT | I_BIT | MODE_SVC @ make sure svc mode
msr  cpsr_c, r0            @ and all irqs disabled
bl    __lookup_processor_type
teq   r10, #0                    @ invalid processor?
moveq     r0, #'p'               @ yes, error 'p'
beq  __error
bl    __lookup_architecture_type
teq   r7, #0                     @ invalid architecture?
moveq     r0, #'a'               @ yes, error 'a'
beq  __error
bl    __create_page_tables
adr   lr, __ret            @ return address
add  pc, r10, #12                @ initialise processor
                           @ (return control reg)

…….

b     SYMBOL_NAME(start_kernel)
```

Linux 内核移植的实例

main.c （2.4 内核）

start_kernel()

- lock_kernel();
- printk(linux_banner); // 打印系统信息
- setup_arch(&command_line); // 初始化系统，setup.c
- printk("Kernel command line: %s\n", saved_command_line);
- parse_options(command_line);
- trap_init();
- init_IRQ();
- sched_init();
- softirq_init();
- time_init();
- console_init();
- init_modules();
- kmem_cache_init();
- calibrate_delay();
- mem_init();
- kmem_cache_sizes_init();

Linux 内核移植的实例

Linux/arch/arm/kernel/setup.c 下的函数 set_arch(char **cmdline_p)， Setup_arch 是特定板子相关的设置函数，命令行字符串和内存起始地址作为参数传递到这个函数， ramdisk 映象的起始地址和结束地址也在这儿更新。 linux/arch/arm 下对每一支持的开发平台有一子目录， MX1 平台为 mach-mx1ads 子目录，存放与平台相关的 .c 文件，同样在 inux/include/asm-arm 下有 arch-mx1ads 子目录，存放与平台相关的 .h 文件。 mach-mx1ads 子目录一般包含 arch.c 、 mm.c 、 irq.c 、 time.c 、 cpu.c 等， arch.c 中包含结构 MACHINE_START ，设置 RAM 的物理地址、 IO 空间物理地址、 IO 虚拟地址、注册 IO map 和 IRQ 初始化函数等。

Linux 内核移植的实例

mmc. 中建立地址的虚实映射关系， irq.c 实现中断控制器的初始化， time.c 和 inux/include/asm-arm/arch-mx1ads/time.h 实现时钟中断和实时时钟处理。 Arch.c 下的 mx1ads_fixup 函数可以对一些参数进行修改，例如命令行字符串、内存起始地址、 ramdisk 映象的起始地址和结束地址等。

上述代码修改完成后可以开始设置中断，中断处理中有两部分比较重要，一是中断控制器，二是时钟中断。 Init_IRQ() 函数初始化中断， linux/arch/arm/mach-mx1ads/irq.c 和 linux/arch/arm/kernel/entry-armv.S 中都包含与硬件体系统结构相关的中断处理函数。大多数系统都使用中断控制器处理中断，而中断控制器与 CPU 的一根中断输入引腿相连，所以必须修改与中断控制器相关的代码，使之与内核的中断处理函数相一致。

linux/include/asm-arm/arch-mx1ads/time.h 中包含了平台相关的时钟代码， ARM　LINUX 内核需要 100Hz 的定时中断。通过设置 MX1　芯片的时钟寄存器可以获得 100Hz 的中断。在时钟中断服务进程中，会调用 do_timer() 函数，该函数对比较器进行写操作从而清除中断标志。

Linux 内核移植的实例

编译环境 (2.4.x)： .config, Makefile,ld 脚本
Make dep
Make menuconfig
Make zImage
Make modules

Make dep 建立文件的依赖关系， make menuconfig 来配置内核（根据各个目录的 config.in 来决定，最开始的 config.in 是最顶层的 Makefile 配置下的 ARCH 目录下的 config.in ），config.in 生成最顶层的 .config, 提供给每个目录下的 Makefile 来决定变异那些文件。

最顶层的 Makefile 决定总体的一下变量，最重要的有：

```
ARCH := arm
…
HOSTCC       = gcc
HOSTCFLAGS   = -Wall -Wstrict-prototypes -O2
-fomit-frame-pointer
…
CROSS_COMPILE  = arm-linux-
```

Linux 内核移植的实例

每个目录下的 Makefile 来决定哪些模块被编译：

```
O_TARGET := misc.o

export-objs                 := mcp-core.o mcp-sa1100.o
ucb1x00-core.o

obj-$(CONFIG_MCP)           += mcp-core.o
obj-$(CONFIG_MCP_SA1100)    += mcp-sa1100.o
obj-$(CONFIG_MCP_UCB1200)   += ucb1x00-core.o
obj-$(CONFIG_MCP_UCB1200_AUDIO) += ucb1x00-
audio.o
obj-$(CONFIG_MCP_UCB1200_TS)  += ucb1x00-ts.o

include $(TOPDIR)/Rules.make
```

Arch/arm/Makefile 也很总要，决定一些总要的参数

```
ifeq ($(CONFIG_CPU_32),y)
PROCESSOR      = armv
TEXTADDR       = 0xC0008000
LDSCRIPT       = arch/arm/vmlinux-armv.lds.in
endif
```

Linux 内核移植的实例

内核的链接： arch/arm/vmlinux-armv(o).lds.in, vmlinux.lds

```
/* ld script to make ARM Linux kernel
 * taken from the i386 version by Russell King
 * Written by Martin Mares <mj@atrey.karlin.mff.cuni.cz>
 */
OUTPUT_ARCH(arm)
ENTRY(stext)
SECTIONS
{
      . = TEXTADDR;
      .init : {                  /* Init code and data        */
            _stext = .;
            __init_begin = .;
                  *(.text.init)
            __proc_info_begin = .;
                  *(.proc.info)
...
           __init_end = .;
      }

      /DISCARD/ : {              /* Exit code and data        */
            *(.text.exit)
            *(.data.exit)
            *(.exitcall.exit)
      }

      .text : {                  /* Real text segment         */
         ...
```

Linux 内核的裁剪

Make menuconfig

文件系统及 init 以后的启动

系统需要一种以结构化格式存储和检索信息的方法，这就需要文件系统的参与。可选的文件系统有： **NFS** ， **RAMdisk** ， **JFFS/JFFS2** 等。为了使用方便，可以选择 **NFS** 做根文件系统，随着开发的深入，可选择 **JFFS/JFFS2** 来在 **Flash** 设备上建立文件系统。只要改变传递给内核的命令行参数“ **root** ＝”（如 **root** ＝ **nsf** ）即可。 **RAMdisk** 是通过将 **RAM** 用作设备来创建和挂装文件系统的一种机制，它通常用于无盘系统（当然包括微型嵌入式设备，它只包含作为永久存储媒质的闪存芯片）。要想能够使用 **RAMdisk** ， **Linux** 内核在配置时必须支持它才行： **RAMdisk support (CONFIG_BLK_DEV_RAM, CONFIG_BLK_DEV_RAM_SIZE, CONFIG_BLK_DEV_INITRD)** 和 **support for the filesystem used (normally ext2 - CONFIG_EXT2_FS)** 。用户可以根据可靠性、健壮性和增强的功能的需求来选择文件系统的类型。

文件系统及 init 以后的启动

内核创建 **kernel_thread**，它执行文件系统中的 **init** 命令（如果在默认的 **/bin**、**/sbin** 或 **/etc** 下没有 **init** 程序，那么内核将执行文件系统的 **/bin** 中的 **shell**）。**init** 程序一般通过执行一些应用程序和启动一些重要的软件来完成系统的启动。

所有的这些行为将会按照它们在 /etc/inittab 中出现的先后次序执行。

总结 **busybox** 的 **init** 启动时依次完成以下任务：

建立 **init** 的 **signal handlers**

初始化 **console(s)**

分析 **inittab** 文件，**/etc/inittab**

运行系统初始化脚本，**busybox** 默认是 **/etc/init.d/rcS**

运行 **inittab** 中所有 wait 类型的命令

运行 **inittab** 中所有 **once** 类型的命令

完成上述过程后，**init** 将循环执行下列任务：

运行 **inittab** 中所有 **respawned** 类型的命令