程序 16-1 linux/tools/build.c

```
/*
 *  linux/tools/build.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * This file builds a disk-image from three different files:
 *
 * - bootsect: max 510 bytes of 8086 machine code, loads the rest
 * - setup: max 4 sectors of 8086 machine code, sets up system parm
 * - system: 80386 code for actual system
 *
 * It does some checking that all files are of the correct type, and
 * just writes the result to stdout, removing headers and padding to
 * the right amount. It also writes some system data to stderr.
 */
/*
 * 该程序从三个不同的程序中创建磁盘映像文件：
 *
 * - bootsect：该文件的 8086 机器码最长为 510 字节，用于加载其他程序。
 * - setup：该文件的 8086 机器码最长为 4 个磁盘扇区，用于设置系统参数。
 * - system：实际系统的 80386 代码。
 *
 * 该程序首先检查所有程序模块的类型是否正确，并将检查结果在终端上显示出来，
 * 然后删除模块头部并扩充大正确的长度。该程序也会将一些系统数据写到 stderr。
 */

/*
 * Changes by tytso to allow root device specification
 *
 * Added swap-device specification: Linux 20.12.91
 */
/*
 * tytso 对该程序作了修改，以允许指定根文件设备。
 *
 * 添加了指定交换设备功能：Linus 20.12.91
 */

#include <stdio.h>      /* fprintf */              // 使用其中的 fprintf()函数。
#include <string.h>                                // 字符串操作函数。
#include <stdlib.h>     /* contains exit */        // 含 exit 函数原型说明。
#include <sys/types.h>  /* unistd.h needs this */  // 该头文件供 unistd.h 文件使用。
#include <sys/stat.h>                              // 含文件状态信息结构定义。
#include <linux/fs.h>                              // 文件系统头文件。
#include <unistd.h>     /* contains read/write */  // 含 read/write 函数原型说明。
#include <fcntl.h>                                 // 包含文件操作模式符号常数。

#define MINIX_HEADER 32             // minix 二进制目标文件模块头部长度为 32 字节。
#define GCC_HEADER 1024             // GCC 头部信息长度为 1024 字节。

#define SYS_SIZE 0x3000             // system 文件最长节数(字节数为 SYS_SIZE*16=128KB)。
```

```
38
   // 默认地把 Linux 根文件系统所在设备设置为在第 2 个硬盘的第 1 个分区上（即设备号为 0x0306），
   // 是因为 Linus 当时开发 Linux 时，把第 1 个硬盘用作 MINIX 系统盘，而第 2 个硬盘用作为 Linux
   // 的根文件系统盘。
39 #define DEFAULT_MAJOR_ROOT 3          // 默认根设备主设备号 - 3（硬盘）。
40 #define DEFAULT_MINOR_ROOT 6          // 默认根设备次设备号 - 6（第 2 个硬盘的第 1 分区）。
41
42 #define DEFAULT_MAJOR_SWAP 0          // 默认交换设备主设备号。
43 #define DEFAULT_MINOR_SWAP 0          // 默认交换设备次设备号。
44
45 /* max nr of sectors of setup: don't change unless you also change
46  * bootsect etc */
   /* 下面指定 setup 模块占的最大扇区数：不要改变该值，除非也改变 bootsect 等相应文件。
47 #define SETUP_SECTS 4                 // setup 最大长度为 4 个扇区（2KB）。
48
49 #define STRINGIFY(x) #x               // 把 x 转换成字符串类型，用于出错显示语句中。
50
   //// 显示出错信息，并终止程序。
51 void die(char * str)
52 {
53         fprintf(stderr,"%s\n",str);
54         exit(1);
55 }
56
   // 显示程序使用方法，并退出。
57 void usage(void)
58 {
59         die("Usage: build bootsect setup system [rootdev] [> image]");
60 }
61
   // 主程序开始。
62 int main(int argc, char ** argv)
63 {
64         int i,c,id;
65         char buf[1024];
66         char major_root, minor_root;
67         char major_swap, minor_swap;
68         struct stat sb;
69
   // 首先检查 build 程序执行时实际命令行参数个数，并根据参数个数作相应设置。如果 build 程序
   // 命令行参数个数不是 4 到 6 个（程序名算作 1 个），则显示程序用法并退出。
70         if ((argc < 4) || (argc > 6))
71                 usage();
   // 若程序命令行上有多于 4 个参数，那么如果根设备名不是软盘（"FLOPPY"），则取该设备文件的
   // 状态信息。若取状态出错则显示信息并退出，否则取该设备名状态结构中的主设备号和次设备号
   // 作为根设备号。如果根设备就是 FLOPPY 设备，则让主设备号和次设备号取 0。表示根设备是当前
   // 启动引导设备。
72         if (argc > 4) {
73                 if (strcmp(argv[4], "FLOPPY")) {
74                         if (stat(argv[4], &sb)) {
75                                 perror(argv[4]);
76                                 die("Couldn't stat root device.");
77                         }
```

```
78                      major_root = MAJOR(sb.st_rdev);   // 取设备名状态结构中设备号。
79                      minor_root = MINOR(sb.st_rdev);
80              } else {
81                      major_root = 0;
82                      minor_root = 0;
83              }
    // 若参数只有 4 个，则让主设备号和次设备号等于系统默认的根设备号。
84      } else {
85              major_root = DEFAULT_MAJOR_ROOT;
86              minor_root = DEFAULT_MINOR_ROOT;
87      }
    // 若程序命令行上有 6 个参数，那么如果最后一个表示交换设备的参数不是无（"NONE"），则取该
    // 设备文件的状态信息。若取状态出错则显示信息并退出，否则取该设备名状态结构中的主设备号
    // 和次设备号作为交换设备号。如果最后一个参数就是"NONE"，则让交换设备的主设备号和次设备
    // 号取为 0。表示交换设备就是当前启动引导设备。
88      if (argc == 6) {
89              if (strcmp(argv[5], "NONE")) {
90                      if (stat(argv[5], &sb)) {
91                              perror(argv[5]);
92                              die("Couldn't stat root device.");
93                      }
94                      major_swap = MAJOR(sb.st_rdev);   // 取设备名状态结构中设备号。
95                      minor_swap = MINOR(sb.st_rdev);
96              } else {
97                      major_swap = 0;
98                      minor_swap = 0;
99              }
    // 若参数没有 6 个而是 5 个，表示命令行上没有带交换设备名。于是就让交换设备主设备号和次设备
    // 号等于系统默认的交换设备号。
100     } else {
101             major_swap = DEFAULT_MAJOR_SWAP;
102             minor_swap = DEFAULT_MINOR_SWAP;
103     }

    // 接下来在标准错误终端上显示上面所选择的根设备主、次设备号和交换设备主、次设备号。如果
    // 主设备号不等于 2（软盘）或 3（硬盘），也不为 0（取系统默认设备），则显示出错信息并退出。
    // 终端的标准输出被定向到文件 Image，因此被用于输出保存内核代码数据，生成内核映像文件。
104     fprintf(stderr, "Root device is (%d, %d)\n", major_root, minor_root);
105     fprintf(stderr, "Swap device is (%d, %d)\n", major_swap, minor_swap);
106     if ((major_root != 2) && (major_root != 3) &&
107         (major_root != 0)) {
108             fprintf(stderr, "Illegal root device (major = %d)\n",
109                     major_root);
110             die("Bad root device --- major #");
111     }
112     if (major_swap && major_swap != 3) {
113             fprintf(stderr, "Illegal swap device (major = %d)\n",
114                     major_swap);
115             die("Bad root device --- major #");
116     }
    // 下面开始执行读取各个文件内容并进行相应的复制处理。首先初始化 1KB 的复制缓冲区，置全 0。
    // 然后以只读方式打开参数 1 指定的文件（bootsect）。从中读取 32 字节的 MINIX 执行文件头结构
    // 内容（参见列表后说明）到缓冲区 buf 中。
```

```
117         for (i=0;i<sizeof buf; i++) buf[i]=0;
118         if ((id=open(argv[1],O_RDONLY,0))<0)
119                 die("Unable to open 'boot'");
120         if (read(id,buf,MINIX_HEADER) != MINIX_HEADER)
121                 die("Unable to read header of 'boot'");
    // 接下来根据 MINIX 头部结构判断 bootsect 是否为一个有效的 MINIX 执行文件。若是，则从文件中
    // 读取 512 字节的引导扇区代码和数据。
    // 0x0301 - MINIX 头部 a_magic 魔数；0x10 - a_flag 可执行；0x04 - a_cpu, Intel 8086 机器码。
122         if (((long *) buf)[0]!=0x04100301)
123                 die("Non-Minix header of 'boot'");
    // 判断头部长度字段 a_hdrlen（字节）是否正确（32 字节）。（后三字节正好没有用，是 0）
124         if (((long *) buf)[1]!=MINIX_HEADER)
125                 die("Non-Minix header of 'boot'");
    // 判断数据段长 a_data 字段(long)内容是否为 0。
126         if (((long *) buf)[3]!=0)
127                 die("Illegal data segment in 'boot'");
    // 判断堆 a_bss 字段(long)内容是否为 0。
128         if (((long *) buf)[4]!=0)
129                 die("Illegal bss in 'boot'");
    // 判断执行点 a_entry 字段(long)内容是否为 0。
130         if (((long *) buf)[5] != 0)
131                 die("Non-Minix header of 'boot'");
    // 判断符号表长字段 a_sym 的内容是否为 0。
132         if (((long *) buf)[7] != 0)
133                 die("Illegal symbol table in 'boot'");
    // 在上述判断都正确的条件下读取文件中随后的实际代码数据，应该返回读取字节数为 512 字节。
    // 因为 bootsect 文件中包含的是 1 个扇区的引导扇区代码和数据，并且最后 2 字节应该是可引导
    // 标志 0xAA55。
134         i=read(id,buf,sizeof buf);
135         fprintf(stderr,"Boot sector %d bytes.\n",i);
136         if (i != 512)
137                 die("Boot block must be exactly 512 bytes");
138         if ((*(unsigned short *)(buf+510)) != 0xAA55)
139                 die("Boot block hasn't got boot flag (0xAA55)");
    // 引导扇区的 506、507 偏移处需存放交换设备号，508、509 偏移处需存放根设备号。
140         buf[506] = (char) minor_swap;
141         buf[507] = (char) major_swap;
142         buf[508] = (char) minor_root;
143         buf[509] = (char) major_root;
    // 然后将该 512 字节的数据写到标准输出 stdout，若写出字节数不对，则显示出错信息并退出。
    // 在 linux/Makefile 中，build 程序标准输出被重定向到内核映像文件名 Image 上，因此引导
    // 扇区代码和数据会被写到 Image 开始的 512 字节处。最后关闭 bootsect 模块文件。
144         i=write(1,buf,512);
145         if (i!=512)
146                 die("Write call failed");
147         close (id);
148
    // 下面以只读方式打开参数 2 指定的文件（setup）。从中读取 32 字节的 MINIX 执行文件头结构
    // 内容到缓冲区 buf 中。处理方式与上面相同。首先以只读方式打开指定的文件 setup。从中读
    // 取 32 字节的 MINIX 执行文件头结构内容到缓冲区 buf 中。
149         if ((id=open(argv[2],O_RDONLY,0))<0)
150                 die("Unable to open 'setup'");
151         if (read(id,buf,MINIX_HEADER) != MINIX_HEADER)
```

```
152                 die("Unable to read header of 'setup'");
    // 接下来根据 MINIX 头部结构判断 setup 是否为一个有效的 MINIX 执行文件。若是，则从文件中
    // 读取 512 字节的引导扇区代码和数据。
    // 0x0301- MINIX 头部 a_magic 魔数；0x10- a_flag 可执行；0x04- a_cpu, Intel 8086 机器码。
153         if (((long *) buf)[0]!=0x04100301)
154                 die("Non-Minix header of 'setup'");
    // 判断头部长度字段 a_hdrlen（字节）是否正确（32 字节）。（后三字节正好没有用，是 0）
155         if (((long *) buf)[1]!=MINIX_HEADER)
156                 die("Non-Minix header of 'setup'");
    // 判断数据段长字段 a_data、堆字段 a_bss、起始执行点字段 a_entry 和符号表字段 a_sym 的内容
    // 是否为 0。必须都为 0。
157         if (((long *) buf)[3]!=0)                       // 数据段长 a_data 字段。
158                 die("Illegal data segment in 'setup'");
159         if (((long *) buf)[4]!=0)                       // 堆 a_bss 字段。
160                 die("Illegal bss in 'setup'");
161         if (((long *) buf)[5] != 0)                     // 执行起始点 a_entry 字段。
162                 die("Non-Minix header of 'setup'");
163         if (((long *) buf)[7] != 0)
164                 die("Illegal symbol table in 'setup'");
    // 在上述判断都正确的条件下读取文件中随后的实际代码数据，并且写到终端标准输出。同时统计
    // 写的长度（i），并在操作结束后关闭 setup 文件。之后判断一下利用 setup 执行写操作的代码
    // 和数据长度值，该值不能大于(SETUP_SECTS * 512)字节，否则就得重新修改 build、bootsect
    // 和 setup 程序中设定的 setup 所占扇区数并重新编译内核。若一切正常就显示 setup 实际长度值。
165         for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c )
166                 if (write(1,buf,c)!=c)
167                         die("Write call failed");
168         close (id);                                     //关闭 setup 模块文件。
169         if (i > SETUP_SECTS*512)
170                 die("Setup exceeds " STRINGIFY(SETUP_SECTS)
171                         " sectors - rewrite build/boot/setup");
172         fprintf(stderr,"Setup is %d bytes.\n",i);
    // 在将缓冲区 buf 清零之后，判断实际写的 setup 长度与(SETUP_SECTS4*512)的数值差，若 setup
    // 长度小于该长度（4*512 字节），则用 NULL 字符将 setup 填足为 4*512 字节。
173         for (c=0 ; c<sizeof(buf) ; c++)
174                 buf[c] = '\0';
175         while (i<SETUP_SECTS*512) {
176                 c = SETUP_SECTS*512-i;
177                 if (c > sizeof(buf))
178                         c = sizeof(buf);
179                 if (write(1,buf,c) != c)
180                         die("Write call failed");
181                 i += c;
182         }
183
    // 下面开始处理 system 模块文件。该文件使用 gas 编译，因此具有 GNU a.out 目标文件格式。
    // 首先以只读方式打开文件，并读取其中 a.out 格式头部结构信息（1KB 长度）。在判断 system
    // 是一个有效的 a.out 格式文件之后，就把该文件随后的所有数据都写到标准输出（Image 文件）
    // 中，并关闭该文件。然后显示 system 模块的长度。若 system 代码和数据长度超过 SYS_SIZE 节
    // （即 128KB 字节），则显示出错信息并退出。若无错，则返回 0，表示正常退出。
184         if ((id=open(argv[3],O_RDONLY,0))<0)
185                 die("Unable to open 'system'");
186         if (read(id,buf,GCC_HEADER) != GCC_HEADER)
187                 die("Unable to read header of 'system'");
```

```
188         if (((long *) buf)[5] != 0)                 // 执行入口点字段 a_entry 值应为 0。
189                 die("Non-GCC header of 'system'");
190         for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c )
191                 if (write(1,buf,c)!=c)
192                         die("Write call failed");
193         close(id);
194         fprintf(stderr,"System is %d bytes.\n",i);
195         if (i > SYS_SIZE*16)
196                 die("System is too big");
197         return(0);
198 }
199
```