第 13 章 建造工具(tools)

13.1 概述

Linux 内核源代码中的 tools 目录中包含一个生成内核磁盘映象文件的工具程序 build.c，该程序将单独编译成可执行文件，在 linux/目录下的 Makefile 文件中被调用运行，用于将所有内核编译代码连接和合并成一个可运行的内核映像文件 image。具体方法是对 boot/中的 bootsect.s、setup.s 使用 8086 汇编器进行编译，分别生成各自的执行模块。再对源代码中的其它所有程序使用 GNU 的编译器 gcc/gas 进行编译，并连接成模块 system。然后使用 build 工具将这三块组合成一个内核映象文件 image。基本编译连接/组合结构如下图 13.1 所示。

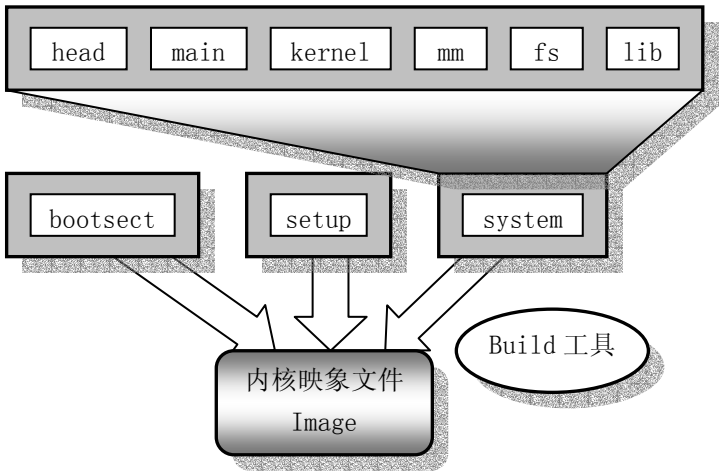

图 13.1 内核编译连接/组合结构

13.2 build.c 程序

13.2.1 功能概述

build 程序使用 4 个参数，分别是 bootsect 文件名、setup 文件名、system 文件名和可选的根文件系统设备文件名。

程序首先检查命令行上最后一个根设备文件名可选参数，若其存在，则读取该设备文件的状态信息结构（stat），取出设备号。若命令行上不带该参数，则使用默认值。

然后对 bootsect 文件进行处理，读取该文件的 minix 执行头部信息，判断其有效性，然后读取随后 512 字节的引导代码数据，判断其是否具有可引导标志 0xAA55，并将前面获取的根设备号写入到 508,509 位移处，最后将该 512 字节代码数据写到 stdout 标准输出，由 Make 文件重定向到 Image 文件。

接下来以类似的方法处理 setup 文件。若该文件长度小于 4 个扇区，则用 0 将其填满为 4 个扇区的长度，并写到标准输出 stdout 中。

最后处理 system 文件。该文件是使用 GCC 编译器产生，所以其执行头部格式是 GCC 类型的，与 linux 定义的 a.out 格式一样。在判断执行入口点是 0 后，就将数据写到标准输出 stdout 中。若其代码数据长度超过 128KB，则显示出错信息。

13.2.2 代码注释

列表 linux/tools/build.c 程序

```
1  /*
2   *  linux/tools/build.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   * This file builds a disk-image from three different files:
9   *
10  * - bootsect: max 510 bytes of 8086 machine code, loads the rest
11  * - setup: max 4 sectors of 8086 machine code, sets up system parm
12  * - system: 80386 code for actual system
13  *
14  * It does some checking that all files are of the correct type, and
15  * just writes the result to stdout, removing headers and padding to
16  * the right amount. It also writes some system data to stderr.
17  */
    /*
     * 该程序从三个不同的程序中创建磁盘映象文件：
     *
     * - bootsect：该文件的 8086 机器码最长为 510 字节，用于加载其它程序。
     * - 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。
     */
18
19 /*
20  * Changes by tytso to allow root device specification
21  */
    /*
     * tytso 对该程序作了修改，以允许指定根文件设备。
     */
22
23 #include <stdio.h>      /* fprintf */            /* 使用其中的 fprintf() */
24 #include <string.h>                              /* 字符串操作 */
25 #include <stdlib.h>     /* contains exit */      /* 含有 exit() */
26 #include <sys/types.h>  /* unistd.h needs this */ /* 供 unistd.h 使用 */
27 #include <sys/stat.h>                            /* 文件状态信息结构 */
```

```
28 #include <linux/fs.h>                            /* 文件系统 */
29 #include <unistd.h>     /* contains read/write */ /* 含有 read()/write() */
30 #include <fcntl.h>                               /* 文件操作模式符号常数 */
31
32 #define MINIX_HEADER 32             // minix 二进制模块头部长度为 32 字节。
33 #define GCC_HEADER 1024             // GCC 头部信息长度为 1024 字节。
34
35 #define SYS_SIZE 0x2000             // system 文件最长节数(字节数为 SYS_SIZE*16=128KB)。
36
37 #define DEFAULT_MAJOR_ROOT 3        // 默认根设备主设备号 - 3（硬盘）。
38 #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）。
39
40 /* max nr of sectors of setup: don't change unless you also change
41  * 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。
42 #define SETUP_SECTS 4               // setup 最大长度为 4 个扇区（4*512 字节）。
43
44 #define STRINGIFY(x) #x             // 用于出错时显示语句中表示扇区数。
45
   //// 显示出错信息，并终止程序。
46 void die(char * str)
47 {
48         fprintf(stderr,"%s\n",str);
49         exit(1);
50 }
51
   // 显示程序使用方法，并退出。
52 void usage(void)
53 {
54         die("Usage: build bootsect setup system [rootdev] [> image]");
55 }
56
57 int main(int argc, char ** argv)
58 {
59         int i,c,id;
60         char buf[1024];
61         char major_root, minor_root;
62         struct stat sb;
63
   // 如果程序命令行参数不是 4 或 5 个，则显示程序用法并退出。
64         if ((argc != 4) && (argc != 5))
65                 usage();
   // 如果参数是 5 个，则说明带有根设备名。
66         if (argc == 5) {
   // 如果根设备名是软盘("FLOPPY")，则取该设备文件的状态信息，若出错则显示信息，退出。
67                 if (strcmp(argv[4], "FLOPPY")) {
68                         if (stat(argv[4], &sb)) {
69                                 perror(argv[4]);
70                                 die("Couldn't stat root device.");
71                         }
   // 若成功则取该设备名状态结构中的主设备号和次设备号。
72                         major_root = MAJOR(sb.st_rdev);
73                         minor_root = MINOR(sb.st_rdev);
```

```
74                  } else {
    // 否则让主设备号和次设备号取 0。
75                          major_root = 0;
76                          minor_root = 0;
77                  }
    // 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。
78          } else {
79                  major_root = DEFAULT_MAJOR_ROOT;
80                  minor_root = DEFAULT_MINOR_ROOT;
81          }
    // 在标准错误终端上显示所选择的根设备主、次设备号。
82          fprintf(stderr, "Root device is (%d, %d)\n", major_root, minor_root);
    // 如果主设备号不等于 2(软盘)或 3(硬盘)，也不等于 0(取系统默认根设备)，则显示出错信息，退出。
83          if ((major_root != 2) && (major_root != 3) &&
84              (major_root != 0)) {
85                  fprintf(stderr, "Illegal root device (major = %d)\n",
86                          major_root);
87                  die("Bad root device --- major #");
88          }
    // 初始化 buf 缓冲区，全置 0。
89          for (i=0;i<sizeof buf; i++) buf[i]=0;
    // 以只读方式打开参数 1 指定的文件(bootsect)，若出错则显示出错信息，退出。
90          if ((id=open(argv[1],O_RDONLY,0))<0)
91                  die("Unable to open 'boot'");
    // 读取文件中的 minix 执行头部信息(参见列表后说明)，若出错则显示出错信息，退出。
92          if (read(id,buf,MINIX_HEADER) != MINIX_HEADER)
93                  die("Unable to read header of 'boot'");
    // 0x0301 - minix 头部 a_magic 魔数；0x10 - a_flag 可执行；0x04 - a_cpu, Intel 8086 机器码。
94          if (((long *) buf)[0]!=0x04100301)
95                  die("Non-Minix header of 'boot'");
    // 判断头部长度字段 a_hdrlen（字节）是否正确。（后三字节正好没有用，是 0）
96          if (((long *) buf)[1]!=MINIX_HEADER)
97                  die("Non-Minix header of 'boot'");
    // 判断数据段长 a_data 字段(long)内容是否为 0。
98          if (((long *) buf)[3]!=0)
99                  die("Illegal data segment in 'boot'");
    // 判断堆 a_bss 字段(long)内容是否为 0。
100         if (((long *) buf)[4]!=0)
101                 die("Illegal bss in 'boot'");
    // 判断执行点 a_entry 字段(long)内容是否为 0。
102         if (((long *) buf)[5] != 0)
103                 die("Non-Minix header of 'boot'");
    // 判断符号表长字段 a_sym 的内容是否为 0。
104         if (((long *) buf)[7] != 0)
105                 die("Illegal symbol table in 'boot'");
    // 读取实际代码数据，应该返回读取字节数为 512 字节。
106         i=read(id,buf,sizeof buf);
107         fprintf(stderr,"Boot sector %d bytes.\n",i);
108         if (i != 512)
109                 die("Boot block must be exactly 512 bytes");
    // 判断 boot 块 0x510 处是否有可引导标志 0xAA55。
110         if ((*(unsigned short *)(buf+510)) != 0xAA55)
111                 die("Boot block hasn't got boot flag (0xAA55)");
```

```
    // 引导块的 508，509 偏移处存放的是根设备号。
112         buf[508] = (char) minor_root;
113         buf[509] = (char) major_root;
    // 将该 boot 块 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，退出。
114         i=write(1,buf,512);
115         if (i!=512)
116                 die("Write call failed");
    // 最后关闭 bootsect 模块文件。
117         close (id);
118
    // 现在开始处理 setup 模块。首先以只读方式打开该模块，若出错则显示出错信息，退出。
119         if ((id=open(argv[2],O_RDONLY,0))<0)
120                 die("Unable to open 'setup'");
    // 读取该文件中的 minix 执行头部信息(32 字节)，若出错则显示出错信息，退出。
121         if (read(id,buf,MINIX_HEADER) != MINIX_HEADER)
122                 die("Unable to read header of 'setup'");
    // 0x0301 - minix 头部 a_magic 魔数；0x10 - a_flag 可执行；0x04 - a_cpu, Intel 8086 机器码。
123         if (((long *) buf)[0]!=0x04100301)
124                 die("Non-Minix header of 'setup'");
    // 判断头部长度字段 a_hdrlen（字节）是否正确。（后三字节正好没有用，是 0）
125         if (((long *) buf)[1]!=MINIX_HEADER)
126                 die("Non-Minix header of 'setup'");
    // 判断数据段长 a_data 字段(long)内容是否为 0。
127         if (((long *) buf)[3]!=0)
128                 die("Illegal data segment in 'setup'");
    // 判断堆 a_bss 字段(long)内容是否为 0。
129         if (((long *) buf)[4]!=0)
130                 die("Illegal bss in 'setup'");
    // 判断执行点 a_entry 字段(long)内容是否为 0。
131         if (((long *) buf)[5] != 0)
132                 die("Non-Minix header of 'setup'");
    // 判断符号表长字段 a_sym 的内容是否为 0。
133         if (((long *) buf)[7] != 0)
134                 die("Illegal symbol table in 'setup'");
    // 读取随后的执行代码数据，并写到标准输出 stdout。
135         for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c )
136                 if (write(1,buf,c)!=c)
137                         die("Write call failed");
    //关闭 setup 模块文件。
138         close (id);
    // 若 setup 模块长度大于 4 个扇区，则算出错，显示出错信息，退出。
139         if (i > SETUP_SECTS*512)
140                 die("Setup exceeds " STRINGIFY(SETUP_SECTS)
141                         " sectors - rewrite build/boot/setup");
    // 在标准错误 stderr 显示 setup 文件的长度值。
142         fprintf(stderr,"Setup is %d bytes.\n",i);
    // 将缓冲区 buf 清零。
143         for (c=0 ; c<sizeof(buf) ; c++)
144                 buf[c] = '\0';
    // 若 setup 长度小于 4*512 字节，则用\0 将 setup 填足为 4*512 字节。
145         while (i<SETUP_SECTS*512) {
146                 c = SETUP_SECTS*512-i;
147                 if (c > sizeof(buf))
```

```
148                         c = sizeof(buf);
149                 if (write(1,buf,c) != c)
150                         die("Write call failed");
151                 i += c;
152         }
153
    // 下面处理 system 模块。首先以只读方式打开该文件。
154         if ((id=open(argv[3],O_RDONLY,0))<0)
155                 die("Unable to open 'system'");
    // system 模块是 GCC 格式的文件，先读取 GCC 格式的头部结构信息(linux 的执行文件也采用该格式)。
156         if (read(id,buf,GCC_HEADER) != GCC_HEADER)
157                 die("Unable to read header of 'system'");
    // 该结构中的执行代码入口点字段 a_entry 值应为 0。
158         if (((long *) buf)[5] != 0)
159                 die("Non-GCC header of 'system'");
    // 读取随后的执行代码数据，并写到标准输出 stdout。
160         for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c )
161                 if (write(1,buf,c)!=c)
162                         die("Write call failed");
    // 关闭 system 文件，并向 stderr 上打印 system 的字节数。
163         close(id);
164         fprintf(stderr,"System is %d bytes.\n",i);
    // 若 system 代码数据长度超过 SYS_SIZE 节（或 128KB 字节），则显示出错信息，退出。
165         if (i > SYS_SIZE*16)
166                 die("System is too big");
167         return(0);
168 }
169
```

13.2.3 相关信息

可执行文件头部数据结构

Minix 可执行文件 a.out 的头部结构如下所示（参见 minix 2.0 源代码 01400 行开始）：

```
struct  exec {
  unsigned char a_magic[2];     // 执行文件魔数。
  unsigned char a_flags;        // 标志（参见后面说明）。
  unsigned char a_cpu;          // cpu 标识号。
  unsigned char a_hdrlen;       // 头部长度。
  unsigned char a_unused;       // 保留给将来使用。
  unsigned short a_version;     // 版本信息（目前未用）。
  long          a_text;         // 代码段长度，字节数。
  long          a_data;         // 数据段长度，字节数。
  long          a_bss;          // 堆长度，字节数。
  long          a_entry;        // 执行入口点地址。
  long          a_total;        // 分配的内存总量。
  long          a_syms;         // 符号表大小。
  ...
};
```

其中标志字段定义为：

```
A_UZP    0x01          // 未映射的 0 页（页数）。
A_PAL    0x02          // 以页边界调整的可执行文件。
A_NSYM   0x04          // 新型符号表。
A_EXEC   0x10          // 可执行文件。
A_SEP    0x20          // 代码和数据是分开的。
```

CPU 标识号为：

```
A_NONE   0x00          // 未知。
A_I8086  0x04          // Intel i8086/8088。
A_M68K   0x0B          // Motorola m68000。
A_NS16K  0x0C          // 国家半导体公司 16032。
A_I80386 0x10          // Intel i80386。
A_SPARC  0x17          // Sun 公司 SPARC。
```

GCC 执行文件头部结构信息参见 linux/include/a.out.h 文件。