

第 11 章 包含文件(include)

















11.1 概述

Linux 0.11 版内核中共有 32 个头文件(*.h)，其中 asm/子目录中含有 4 个，linux/子目录中含有 10 个，sys/子目录中含有 5 个。从下一节开始我们首先描述 include/目录下的 13 个头文件，然后依次说明每个子目录中的文件。说明顺序按照文件名称排序进行。

11.2 include/目录下的文件

该目录下的文件列表如下（列表 11.1）：

列表 11.1 linux/include/目录下的文件

	Name	Size	Last modified (GMT)	Description
	asm/		1991-09-17 13:08:31	
	linux/		1991-11-02 13:35:49	
	sys/		1991-09-17 15:06:07	
	a.out.h	6047 bytes	1991-09-17 15:10:49	m
	const.h	321 bytes	1991-09-17 15:12:39	m
	ctype.h	1049 bytes	1991-11-07 17:30:47	m
	errno.h	1268 bytes	1991-09-17 15:04:15	m
	fcntl.h	1374 bytes	1991-09-17 15:12:39	m
	signal.h	1762 bytes	1991-09-22 19:58:04	m
	stdarg.h	780 bytes	1991-09-17 15:02:23	m
	stddef.h	286 bytes	1991-09-17 15:02:17	m
	string.h	7881 bytes	1991-09-17 15:04:09	m
	termios.h	5325 bytes	1991-11-25 20:02:08	m
	time.h	734 bytes	1991-09-17 15:02:02	m
	unistd.h	6410 bytes	1991-11-25 20:18:55	m
	utime.h	225 bytes	1991-09-17 15:03:38	m

11.3 a.out.h 文件

11.3.1 功能描述

a.out.h 头文件主要定义了二进制执行文件 a.out (Assembly out) 的格式。其中包括三个数据结构和一些宏函数。

11.3.2 代码注释

列表 11.2 linux/include/a.out.h 文件

```

1 #ifndef A_OUT_H
2 #define A_OUT_H
3
4 #define GNU_EXEC_MACROS
5
6 // 执行文件结构。
7 // =====
8 // unsigned long a_magic      // 执行文件魔数。使用 N_MAGIC 等宏访问。
9 // unsigned a_text           // 代码长度，字节数。
10 // unsigned a_data           // 数据长度，字节数。
11 // unsigned a_bss            // 文件中的未初始化数据区长度，字节数。
12 // unsigned a_syms           // 文件中的符号表长度，字节数。
13 // unsigned a_entry          // 执行开始地址。
14 // unsigned a_trsize         // 代码重定位信息长度，字节数。
15 // unsigned a_drsize         // 数据重定位信息长度，字节数。
16 // -----
17 struct exec {
18     unsigned long a_magic;      /* Use macros N_MAGIC, etc for access */
19     unsigned a_text;           /* length of text, in bytes */
20     unsigned a_data;           /* length of data, in bytes */
21     unsigned a_bss;            /* length of uninitialized data area for file, in bytes */
22     unsigned a_syms;           /* length of symbol table data in file, in bytes */
23     unsigned a_entry;          /* start address */
24     unsigned a_trsize;         /* length of relocation info for text, in bytes */
25     unsigned a_drsize;         /* length of relocation info for data, in bytes */
26 };
27
28 // 用于取执行结构中的魔数。
29 #ifndef N_MAGIC
30 #define N_MAGIC(exec) ((exec).a_magic)
31 #endif
32
33 #ifndef OMAGIC
34 /* Code indicating object file or impure executable. */
35 /* 指明为目标文件或者不纯的可执行文件的代号 */
36 #define OMAGIC 0407
37 /* Code indicating pure executable. */
38 /* 指明为纯可执行文件的代号 */
39 #define NMAGIC 0410
40 /* Code indicating demand-paged executable. */
41 /* 指明为需求分页处理的可执行文件 */
42 #define ZMAGIC 0413
43 #endif /* not OMAGIC */
44
45 // 如果魔数不能被识别，则返回真。
46 #ifndef N_BADMAG
47 #define N_BADMAG(x) \
48     (N_MAGIC(x) != OMAGIC && N_MAGIC(x) != NMAGIC \
49     && N_MAGIC(x) != ZMAGIC)
50 #endif

```

```

35
36 #define _N_BADMAG(x) \
37 (N_MAGIC(x) != OMAGIC && N_MAGIC(x) != NMAGIC \
38  && N_MAGIC(x) != ZMAGIC)
39
// 程序头在内存中的偏移位置。
40 #define N_HDROFF(x) (SEGMENT_SIZE - sizeof (struct exec))
41
// 代码起始偏移值。
42 #ifndef N_TXTOFF
43 #define N_TXTOFF(x) \
44 (N_MAGIC(x) == ZMAGIC ? N_HDROFF((x)) + sizeof (struct exec) : sizeof (struct exec))
45 #endif
46
// 数据起始偏移值。
47 #ifndef N_DATOFF
48 #define N_DATOFF(x) (N_TXTOFF(x) + (x).a_text)
49 #endif
50
// 代码重定位信息偏移值。
51 #ifndef N_TRELOFF
52 #define N_TRELOFF(x) (N_DATOFF(x) + (x).a_data)
53 #endif
54
// 数据重定位信息偏移值。
55 #ifndef N_DRELOFF
56 #define N_DRELOFF(x) (N_TRELOFF(x) + (x).a_trsize)
57 #endif
58
// 符号表偏移值。
59 #ifndef N_SYMOFF
60 #define N_SYMOFF(x) (N_DRELOFF(x) + (x).a_drsize)
61 #endif
62
// 字符串信息偏移值。
63 #ifndef N_STROFF
64 #define N_STROFF(x) (N_SYMOFF(x) + (x).a_syms)
65 #endif
66
67 /* Address of text segment in memory after it is loaded. */
68 /* 代码段加载到内存中后的地址 */
69 #ifndef N_TXTADDR
70 #define N_TXTADDR(x) 0
71 #endif
72 /* Address of data segment in memory after it is loaded.
73  Note that it is up to you to define SEGMENT_SIZE
74  on machines not listed here. */
75 /* 数据段加载到内存中后的地址。
76 注意，对于下面没有列出名称的机器，需要你自己来定义
77 对应的 SEGMENT_SIZE */
78 #if defined(vax) || defined(hp300) || defined(pyr)
79 #define SEGMENT_SIZE PAGE_SIZE

```

```

77 #endif
78 #ifdef hp300
79 #define PAGE_SIZE 4096
80 #endif
81 #ifdef sony
82 #define SEGMENT_SIZE 0x2000
83 #endif /* Sony. */
84 #ifdef is68k
85 #define SEGMENT_SIZE 0x20000
86 #endif
87 #if defined(m68k) && defined(PORTAR)
88 #define PAGE_SIZE 0x400
89 #define SEGMENT_SIZE PAGE_SIZE
90 #endif
91
92 #define PAGE_SIZE 4096
93 #define SEGMENT_SIZE 1024
94
95 // 以段为界的大小。
96 #define N_SEGMENT_ROUND(x) (((x) + SEGMENT_SIZE - 1) & ~(SEGMENT_SIZE - 1))
97
98 // 代码段尾地址。
99 #define N_TXTENDADDR(x) (N_TXTADDR(x) + (x).a_text)
100
101 // 数据开始地址。
102 #ifndef N_DATADDR
103 #define N_DATADDR(x) \
104     (N_MAGIC(x) == OMAGIC ? (N_TXTENDADDR(x)) \
105      : (N_SEGMENT_ROUND(N_TXTENDADDR(x))))
106 #endif
107
108 /* Address of bss segment in memory after it is loaded. */
109 /* bss 段加载到内存以后的地址 */
110 #ifndef N_BSSADDR
111 #define N_BSSADDR(x) (N_DATADDR(x) + (x).a_data)
112 #endif
113
114 // nlist 结构。
115 #ifndef N_LIST_DECLARED
116 struct nlist {
117     union {
118         char *n_name;
119         struct nlist *n_next;
120         long n_strx;
121     } n_un;
122     unsigned char n_type;
123     char n_other;
124     short n_desc;
125     unsigned long n_value;
126 };
127 #endif
128
129 // 下面定义 exec 结构中的变量偏移值。

```

```

124 #ifndef N\_UNDF
125 #define N\_UNDF 0
126 #endif
127 #ifndef N\_ABS
128 #define N\_ABS 2
129 #endif
130 #ifndef N\_TEXT
131 #define N\_TEXT 4
132 #endif
133 #ifndef N\_DATA
134 #define N\_DATA 6
135 #endif
136 #ifndef N\_BSS
137 #define N\_BSS 8
138 #endif
139 #ifndef N\_COMM
140 #define N\_COMM 18
141 #endif
142 #ifndef N\_FN
143 #define N\_FN 15
144 #endif
145
146 #ifndef N\_EXT
147 #define N\_EXT 1
148 #endif
149 #ifndef N\_TYPE
150 #define N\_TYPE 036
151 #endif
152 #ifndef N\_STAB
153 #define N\_STAB 0340
154 #endif
155
156 /* The following type indicates the definition of a symbol as being
157    an indirect reference to another symbol. The other symbol
158    appears as an undefined reference, immediately following this symbol.
159
160    Indirection is asymmetrical. The other symbol's value will be used
161    to satisfy requests for the indirect symbol, but not vice versa.
162    If the other symbol does not have a definition, libraries will
163    be searched to find a definition. */
164 /* 下面的类型指明了符号的定义作为对另一个符号的间接引用。紧接该符号的其它
165    * 的符号呈现为未定义的引用。
166    *
167    * 间接性是不对称的。其它符号的值将被用于满足间接符号的请求，但反之不然。
168    * 如果其它符号并没有定义，则将搜索库来寻找一个定义 */
164 #define N\_INDR 0xa
165
166 /* The following symbols refer to set elements.
167    All the N\_SET\[ATDB\] symbols with the same name form one set.
168    Space is allocated for the set in the text section, and each set
169    element's value is stored into one word of the space.
170    The first word of the space is the length of the set (number of elements).
171

```

```

172 The address of the set is made into an N_SETV symbol
173 whose name is the same as the name of the set.
174 This symbol acts like a N_DATA global symbol
175 in that it can satisfy undefined external references. */
/* 下面的符号与集合元素有关。所有具有相同名称 N_SET[ATDB] 的符号
   形成一个集合。在代码部分中已为集合分配了空间，并且每个集合元素
   的值存放在一个字（word）的空间。空间的第一个字存有集合的长度（集合元素数目）。

```

集合的地址被放入一个 N_SETV 符号，它的名称与集合同名。

在满足未定义的外部引用方面，该符号的行为象一个 N_DATA 全局符号。*/

```

176
177 /* These appear as input to LD, in a .o file. */
/* 以下这些符号在目标文件中是作为链接程序 LD 的输入。*/
178 #define N_SETA 0x14 /* Absolute set element symbol */
                        /* 绝对集合元素符号 */
179 #define N_SETT 0x16 /* Text set element symbol */
                        /* 代码集合元素符号 */
180 #define N_SETD 0x18 /* Data set element symbol */
                        /* 数据集合元素符号 */
181 #define N_SETB 0x1A /* Bss set element symbol */
                        /* Bss 集合元素符号 */
182
183 /* This is output from LD. */
/* 下面是 LD 的输出。*/
184 #define N_SETV 0x1C /* Pointer to set vector in data area. */
                        /* 指向数据区中集合向量。*/
185
186 #ifndef N_RELOCATION_INFO_DECLARED
187
188 /* This structure describes a single relocation to be performed.
189 The text-relocation section of the file is a vector of these structures,
190 all of which apply to the text section.
191 Likewise, the data-relocation section applies to the data section. */
/* 下面的结构描述执行一个重定位的操作。
   文件的代码重定位部分是这些结构的一个向量，所有这些适用于代码部分。
   类似地，数据重定位部分适用于数据部分。*/
192
193 // 重定位信息结构。
194 struct relocation_info
195 {
196     /* Address (within segment) to be relocated. */
        /* 需要重定位的地址（在段内）。*/
197     int r_address;
198     /* The meaning of r_symbolnum depends on r_extern. */
        /* r_symbolnum 的含义与 r_extern 有关。*/
199     unsigned int r_symbolnum:24;
200     /* Nonzero means value is a pc-relative offset
        and it should be relocated for changes in its own address
201     as well as for changes in the symbol or section specified. */
        /* 非零意味着值是一个 pc 相关的偏移值，因而需要被重定位到自己
        的地址处以及符号或节指定的改变。*/ [??]
202     unsigned int r_pcrel:1;
203     /* Length (as exponent of 2) of the field to be relocated.

```

```

204      Thus, a value of 2 indicates 1<<2 bytes.  */
      /* 需要被重定位的字段长度（是 2 的次方）。
         因此，若值是 2 则表示 1<<2 字节数。*/
205 unsigned int r_length:2;
206 /* 1 => relocate with value of symbol.
207      r_symbolnum is the index of the symbol
208      in file's the symbol table.
209      0 => relocate with the address of a segment.
210      r_symbolnum is N_TEXT, N_DATA, N_BSS or N_ABS
211      (the N_EXT bit may be set also, but signifies nothing).  */
      /* 1 => 以符号的值重定位。
         r_symbolnum 是文件符号表中符号的索引。
         0 => 以段的地址进行重定位。
         r_symbolnum 是 N_TEXT、N_DATA、N_BSS 或 N_ABS
         (N_EXT 比特位也可以被设置，但是毫无意义)。*/
212 unsigned int r_extern:1;
213 /* Four bits that aren't used, but when writing an object file
214      it is desirable to clear them.  */
      /* 没有使用的 4 个比特位，但是当进行写一个目标文件时
         最好将它们复位掉。*/
215 unsigned int r_pad:4;
216 };
217 #endif /* no N_RELOCATION_INFO_DECLARED.  */
218
219
220 #endif /* __A_OUT_GNU_H__  */
221

```

11.3.3 其它信息

a.out 执行文件格式

Linux 内核 0.11 版仅支持 a.out (Assembly out) 执行文件格式，虽然这种格式目前已经渐渐不用，而使用功能更为齐全的 ELF (Executable and Link Format) 格式，但是由于其简单性，作为学习入门的材料正好比较适用。下面全面介绍一下 a.out 格式。

在头文件<a.out.h>中声明了三个数据结构以及一些宏函数。这些数据结构描述了系统上可执行的机器码文件（二进制文件）。

一个执行文件共可有七个部分（七节）组成。按照顺序，这些部分是：

执行头部分(exec header)

执行文件头部分。该部分中含有一些参数，内核使用这些参数将执行文件加载到内存中并执行，而链接程序(ld)使用这些参数将一些二进制目标文件组合成一个可执行文件。这是唯一必要的组成部分。

代码段部分(text segment)

含有程序执行使被加载到内存中的指令代码和相关数据。可以以只读形式进行加载。

数据段部分(data segment)

这部分含有已经初始化过的数据，总是被加载到可读写的内存中。

代码重定位部分(text relocations)

这部分含有供链接程序使用的记录数据。在组合二进制目标文件时用于定位代码段中的指针或地址。

数据重定位部分(data relocations)

与代码重定位部分的作用类似，但是是用于数据段中指针的重定位。

符号表部分(symbol table)

这部分同样含有供链接程序使用的记录数据，用于在二进制目标文件之间对命名的变量和函数（符号）进行交叉引用。

字符串表部分(string table)

该部分含有与符号名相对应的字符串。

每个二进制执行文件均以执行数据结构（exec structure）开始。该数据结构的形式如下：

```
struct exec {
    unsigned long a_midmag;
    unsigned long a_text;
    unsigned long a_data;
    unsigned long a_bss;
    unsigned long a_syms;
    unsigned long a_entry;
    unsigned long a_trsize;
    unsigned long a_drsize;
};
```

各个字段的功能如下：

在 出 执 段 写内存 据 而	a_midmag	该字段含有被 N_GETFLAG()、N_GETMID 和 N_GETMAGIC() 访问的子部分，是由链接程序运行时加载到进程地址空间。宏 N_GETMID() 用于返回机器标识符(machine-id)，指示二进制文件将在什么机器上运行。N_GETMAGIC() 宏指明魔数，它唯一地确定了二进制执行文件与其它加载的文件之间的区别。字段中必须包含以下值之一：
	OMAGIC	表示代码和数据段紧随在执行头后面并且是连续存放的。内核将代码和数据都加载到可读写内存中。
	NMAGIC	同 OMAGIC 一样，代码和数据段紧随在执行头后面并且是连续存放的。然而内核将代码加载到了只读内存中，并把数据段加载到代码段后下一页可读边界开始。
	ZMAGIC	内核在必要时从二进制执行文件中加载独立的页面。执行头部、代码段和数据段都被链接程序处理成多个页面大小的块。内核加载的代码页面时只读的，数据段的页面是可写的。
	a_text	该字段含有代码段的长度值，字节数。
	a_data	该字段含有数据段的长度值，字节数。

a_bss	含有‘bss 段’的长度，内核用其设置在数据段后初始的 break (brk)。内核在加载程序时，这段可写内存显现出处于数据段后面，并且初始时为全零。
a_syms	含有符号表部分的字节长度值。
a_entry	含有内核将执行文件加载到内存中以后，程序执行起始点的内存地址。
a_trsize	该字段含有代码重定位表的大小，是字节数。
a_drsize	该字段含有数据重定位表的大小，是字节数。

在 a.out.h 头文件中定义了几个宏，这些宏使用 exec 结构来测试一致性或者定位执行文件中各个部分（节）的位置偏移值。这些宏有：

N_BADMAG(exec) 如果 a_magic 字段不能被识别，则返回非零值。

N_TXTOFF(exec) 代码段的起始位置字节偏移值。

N_DATOFF(exec) 数据段的起始位置字节偏移值。

N_DRELOFF(exec) 数据重定位表的起始位置字节偏移值。

N_TRELOFF(exec) 代码重定位表的起始位置字节偏移值。

N_SYMOFF(exec) 符号表的起始位置字节偏移值。

N_STROFF(exec) 字符串表的起始位置字节偏移值。

重定位记录具有标准格式，它使用重定位信息(relocation_info)结构来描述：

```
struct relocation_info {
    int            r_address;
    unsigned int   r_symbolnum : 24,
                  r_pcrel : 1,
                  r_length : 2,
                  r_extern : 1,
                  r_baserel : 1,
                  r_jmptable : 1,
                  r_relative : 1,
                  r_copy : 1;
};
```

该结构中各字段的含义如下：

r_address 该字段含有需要链接程序处理（编辑）的指针的字节偏移值。代码重定位的偏移值是从代码段开始处计数的，数据重定位的偏移值是从数据段开始处计算的。链接程序会将已经存储在该偏移处的值与使用重定位记录计算出的新值相加。

r_symbolnum 该字段含有符号表中一个符号结构的序号值（不是字节偏移值）。链接程序在算出符号的绝对地址以后，就将该地址加到正在进行重定位的指针上。（如果 **r_extern** 比特位是 0，那么情况就不同，见下面。）

r_pcrel 如果设置了该位，链接程序就认为正在更新一个指针，该指针使用 pc 相关寻址方式，是属于机器码指令部分。当运行程序使用这个被重定位的指针时，该指针的地址被隐式地加到该指针上。

r_length 该字段含有指针长度的 2 的次方值：0 表示 1 字节长，1 表示 2 字节长，2 表示 4 字节长。

r_extern 如果被置位，表示该重定位需要一个外部引用；此时链接程序必须使用一个符号地址来更新相应指针。当该位是 0 时，则重定位是“局部”的；链接程序更新指针以反映各个段加载地址中的变化，而不是反映一个符号值的变化（除非同时设置了 **r_baserel**，见下面）。在这种情况下，**r_symbolnum** 字段的内容是一个 **n_type** 值（见下面）；这类字段告诉链接程序被重定位的指针指向那个段。

r_baserel 如果设置了该位，则 **r_symbolnum** 字段指定的符号将被重定位成全局偏移表 (Global Offset Table) 中的一个偏移值。在运行时刻，全局偏移表该偏移处被设置为符号的地址。

r_jmptable 如果被置位，则 **r_symbolnum** 字段指定的符号将被重定位成过程链接表 (Procedure Linkage Table) 中的一个偏移值。

r_relative 如果被置位，则说明此重定位与该目标文件将成为其组成部分的映象文件在运行时被加载的地址相关。这类重定位仅在共享目标文件中出现。

r_copy 如果被置位，该重定位记录指定了一个符号，该符号的内容将被复制到 **r_address** 指定的地方。该复制操作是通过共享目标模块中一个合适的数据项中的运行时刻链接程序完成的。

符号将名称映射为地址（或者更通俗地讲是字符串映射到值）。由于链接程序对地址的调整，一个符号的名称必须用来表示其地址，直到已被赋予一个绝对地址值。符号是由符号表中固定长度的记录以及字符串表中的可变长度名称组成。符号表是 **nlist** 结构的一个数组，如下所示：

```
struct nlist {
    union {
        char    *n_name;
        long    n_strx;
    } n_un;
    unsigned char n_type;
    char         n_other;
```

```

        short          n_desc;
        unsigned long   n_value;
    };

```

其中各字段的含义为：

`n_un.n_strx` 含有本符号的名称在字符串表中的字节偏移值。当程序使用 `nlist()` 函数访问一个符号表时，该字段被替换为 `n_un.n_name` 字段，这是内存中字符串的指针。

`n_type` 用于链接程序确定如何更新符号的值。使用位屏蔽(bitmasks)可以将 `n_type` 字段分割成三个子字段，对于 `N_EXT` 类型位置位的符号，链接程序将它们看作是“外部的”符号，并且允许其它二进制目标文件对它们的引用。`N_TYPE` 屏蔽码用于链接程序感兴趣的比特位：

`N_UNDF` 一个未定义的符号。链接程序必须在其它二进制目标文件中定位一个具有相名称的外部符号，以确定该符号的绝对数据值。特殊情况下，如果 `n_type` 字段是非零值，并且没有二进制文件定义了这个符号，则链接程序在 BSS 段中将该符号解析地址，保留长度等于 `n_value` 的字节。如果符号在多于一个二进制目标文件中都定义并且这些二进制目标文件对其长度值不一致，则链接程序将选择所有二进制目标文件中最大的长度。

`N_ABS` 一个绝对符号。链接程序不会更新一个绝对符号。

`N_TEXT` 一个代码符号。该符号的值是代码地址，链接程序在合并二进制目标文件时更新其值。

`N_DATA` 一个数据符号；与 `N_TEXT` 类似，但是用于数据地址。对应代码和数据符号的值不是文件的偏移值而是地址；为了找出文件的偏移，就有必要确定相关部分开的地址并减去它，然后加上该部分的偏移。

`N_BSS` 一个 BSS 符号；与代码或数据符号类似，但在二进制目标文件中没有对应的移。

`N_FN` 一个文件名符号。在合并二进制目标文件时，链接程序会将该符号插入在二进制文件中的符号之前。符号的名称就是给予链接程序的文件名，而其值是二进制文件首个代码段地址。链接和加载时不需要文件名符号，但对于调式程序非常有用。

`N_STAB` 屏蔽码用于选择符号调式程序(例如 `gdb`)感兴趣的位；其值在 `stab()` 中说明。

`n_other` 该字段按照 `n_type` 确定的段，提供有关符号重定位操作的符号独立性信息。目前，`n_other` 字段的最低 4 位含有两个值之一：`AUX_FUNC` 和 `AUX_OBJECT` (有关定义参见 `<link.h>`)。`AUX_FUNC` 将符号与可调用的函数相关，`AUX_OBJECT` 将符号与数据相关，而不管它们是位于代码段还是数据段。该字段主要用于链接程序 `ld`，用于动态可执行程序创建。

`n_desc` 保留给调式程序使用；链接程序不对其进行处理。不同的调试程序将该字段用作不同

的用途。

`n_value` 含有符号的值。对于代码、数据和 BSS 符号，这是一个地址；对于其它符号（例如调式程序符号），值可以是任意的。

字符串表是由长度为 `u_int32_t` 后跟一 `null` 结尾的符号字符串组成。长度代表整个表的字节大小，所以在 32 位的机器上其最小值（或者是第 1 个字符串的偏移）总是 4。

11.4 const.h 文件

11.4.1 功能描述

该文件中定义了 `i` 节点中文件属性和类型 `i_mode` 字段所用到的一些标志位常量符号。

11.4.2 代码注释

列表 linux/include/const.h 文件

```

1 #ifndef CONST\_H
2 #define CONST\_H
3
4 #define BUFFER\_END 0x200000          // 定义缓冲使用内存的末端(代码中没有使用该常量)。
5
6 // i 节点数据结构中 i_mode 字段的各标志位。
7 #define I\_TYPE          0170000      // 指明 i 节点类型。
8 #define I\_DIRECTORY    0040000      // 是目录文件。
9 #define I\_REGULAR       0100000      // 常规文件，不是目录文件或特殊文件。
10 #define I\_BLOCK\_SPECIAL 0060000     // 块设备特殊文件。
11 #define I\_CHAR\_SPECIAL  0020000     // 字符设备特殊文件。
12 #define I\_NAMED\_PIPE    0010000     // 命名管道。
13 #define I\_SET\_UID\_BIT    0004000     // 在执行时设置有效用户 id 类型。
14 #define I\_SET\_GID\_BIT    0002000     // 在执行时设置有效组 id 类型。
15 #endif
16

```

11.5 ctype.h 文件

11.5.1 功能描述

该文件定义了一些有关字符类型判断和转换的宏，是使用数组（表）进行操作的。当使用宏时，字符是作为一个表（`__ctype`）中的索引，从表中获取一个字节，于是可得到相关的比特位。

11.5.2 代码注释

列表 linux/include/ctype.h 文件

```

1 #ifndef CTYPE_H
2 #define CTYPE_H
3
4 #define U      0x01    /* upper */           // 该比特位用于大写字符[A-Z]。
5 #define L      0x02    /* lower */           // 该比特位用于小写字符[a-z]。
6 #define D      0x04    /* digit */           // 该比特位用于数字[0-9]。
7 #define C      0x08    /* cntrl */           // 该比特位用于控制字符。
8 #define P      0x10    /* punct */           // 该比特位用于标点字符。
9 #define S      0x20    /* white space (space/lf/tab) */ // 用于空白字符，如空格、\t、\n 等。
10 #define X      0x40    /* hex digit */        // 该比特位用于十六进制数字。
11 #define SP     0x80    /* hard space (0x20) */   // 该比特位用于空格字符(0x20)。
12
13 extern unsigned char ctype[];                // 字符特性数组(表)，定义了各个字符对应上面的属性。
14 extern char ctmp;                             // 一个临时字符变量(在 fs/ctype.c 中定义)。
15
16 // 下面是一些确定字符类型的宏。
17 #define isalnum(c) ((ctype+1)[c]&(U|L|D))      // 是字符或数字[A-Z]、[a-z]或[0-9]。
18 #define isalpha(c) ((ctype+1)[c]&(U|L))          // 是字符。
19 #define iscntrl(c) ((ctype+1)[c]&(C))            // 是控制字符。
20 #define isdigit(c) ((ctype+1)[c]&(D))            // 是数字。
21 #define isgraph(c) ((ctype+1)[c]&(P|U|L|D))      // 是图形字符。
22 #define islower(c) ((ctype+1)[c]&(L))            // 是小写字符。
23 #define isprint(c) ((ctype+1)[c]&(P|U|L|D|SP)) // 是可打印字符。
24 #define ispunct(c) ((ctype+1)[c]&(P))            // 是标点符号。
25 #define isspace(c) ((ctype+1)[c]&(S))            // 是空白字符如空格、\f、\n、\r、\t、\v。
26 #define isupper(c) ((ctype+1)[c]&(U))            // 是大写字符。
27 #define isxdigit(c) ((ctype+1)[c]&(D|X))          // 是十六进制数字。
28
29 #define isascii(c) (((unsigned) c)<=0x7f)          // 是 ASCII 字符。
30 #define toascii(c) (((unsigned) c)&0x7f)           // 转换成 ASCII 字符。
31
32 #define tolower(c) (ctmp=c, isupper(ctmp)? ctmp-'A'+a: ctmp) // 转换成对应小写字符。
33 #define toupper(c) (ctmp=c, islower(ctmp)? ctmp-'a'+A: ctmp) // 转换成对应大写字符。
34
35 #endif

```

11.6 errno.h 文件

11.6.1 功能描述

11.6.2 代码注释

列表 linux/include/errno.h 文件

```

1 #ifndef ERRNO_H

```

```

2 #define ERRNO_H
3
4 /*
5  * ok, as I hadn't got any other source of information about
6  * possible error numbers, I was forced to use the same numbers
7  * as minix.
8  * Hopefully these are posix or something. I wouldn't know (and posix
9  * isn't telling me - they want $$$ for their f***ing standard).
10 *
11 * We don't use the _SIGN cludge of minix, so kernel returns must
12 * see to the sign by themselves.
13 *
14 * NOTE! Remember to change strerror() if you change this file!
15 */
/*
* ok, 由于我没有得到任何其它有关出错号的资料, 我只能使用与 minix 系统
* 相同的出错号了。
* 希望这些是 POSIX 兼容的或者在一定程度上是这样的, 我不知道 (而且 POSIX
* 没有告诉我 - 要获得他们的混蛋标准需要出钱)。
*
* 我们没有使用 minix 那样的 _SIGN 簇, 所以内核的返回值必须自己辨别正负号。
*
* 注意! 如果你改变该文件的话, 记着也要修改 strerror() 函数。
*/
16
17 extern int errno;
18
19 #define ERROR          99          // 一般错误。
20 #define EPERM          1          // 操作没有许可。
21 #define ENOENT         2          // 文件或目录不存在。
22 #define ESRCH          3          // 指定的进程不存在。
23 #define EINTR          4          // 中断的函数调用。
24 #define EIO            5          // 输入/输出错。
25 #define ENXIO          6          // 指定设备或地址不存在。
26 #define E2BIG          7          // 参数列表太长。
27 #define ENOEXEC        8          // 执行程序格式错误。
28 #define EBADF          9          // 文件句柄(描述符)错误。
29 #define ECHILD        10         // 子进程不存在。
30 #define EAGAIN         11         // 资源暂时不可用。
31 #define ENOMEM        12         // 内存不足。
32 #define EACCES        13         // 没有许可权限。
33 #define EFAULT        14         // 地址错。
34 #define ENOTBLK       15         // 不是块设备文件。
35 #define EBUSY         16         // 资源正忙。
36 #define EEXIST        17         // 文件已存在。
37 #define EXDEV         18         // 非法连接。
38 #define ENODEV        19         // 设备不存在。
39 #define ENOTDIR       20         // 不是目录文件。
40 #define EISDIR        21         // 是目录文件。
41 #define EINVAL        22         // 参数无效。
42 #define ENFILE        23         // 系统打开文件数太多。
43 #define EMFILE        24         // 打开文件数太多。
44 #define ENOTTY        25         // 不恰当的 IO 控制操作(没有 tty 终端)。

```

```

45 #define ETXTBSY      26          // 不再使用。
46 #define EFBIG        27          // 文件太大。
47 #define ENOSPC       28          // 设备已满（设备已经没有空间）。
48 #define EPIPE       29          // 无效的文件指针重定位。
49 #define EROFS        30          // 文件系统只读。
50 #define EMLINK       31          // 连接太多。
51 #define EPIPE       32          // 管道错。
52 #define EDOM         33          // 域(domain)出错。
53 #define ERANGE       34          // 结果太大。
54 #define EDEADLK      35          // 避免资源死锁。
55 #define ENAMETOOLONG 36          // 文件名太长。
56 #define ENOLCK       37          // 没有锁定可用。
57 #define ENOSYS       38          // 功能还没有实现。
58 #define ENOTEMPTY    39          // 目录不空。
59
60 #endif
61

```

11.7 fcntl.h 文件

11.10.1 功能描述

11.7.2 代码注释

列表 linux/include/fcntl.h 文件

```

1 #ifndef FCNTL_H
2 #define FCNTL_H
3
4 #include <sys/types.h>    // 类型头文件。定义了基本的系统数据类型。
5
6 /* open/fcntl - NOCTTY, NDELAY isn't implemented yet */
7 /* open/fcntl - NOCTTY 和 NDELAY 现在还没有实现 */
8 #define O_ACCMODE      00003    // 文件访问模式屏蔽码。
9 // 打开文件 open() 和文件控制 fcntl() 函数使用的文件访问模式。同时只能使用三者之一。
10 #define O_RDONLY       00        // 以只读方式打开文件。
11 #define O_WRONLY       01        // 以只写方式打开文件。
12 #define O_RDWR        02        // 以读写方式打开文件。
13 // 下面是文件创建标志，用于 open()。可与上面访问模式用'位或'的方式一起使用。
14 #define O_CREAT        00100    /* not fcntl */ // 如果文件不存在就创建。
15 #define O_EXCL         00200    /* not fcntl */ // 独占使用文件标志。
16 #define O_NOCTTY       00400    /* not fcntl */ // 不分配控制终端。
17 #define O_TRUNC        01000    /* not fcntl */ // 若文件已存在且是写操作，则长度截为 0。
18 #define O_APPEND       02000    // 以添加方式打开，文件指针置为文件尾。
19 #define O_NONBLOCK     04000    /* not fcntl */ // 非阻塞方式打开和操作文件。
20 #define O_NDELAY       O_NONBLOCK // 非阻塞方式打开和操作文件。
21
22 /* Defines for fcntl-commands. Note that currently
23  * locking isn't supported, and other things aren't really

```

```

21  * tested.
22  */
/* 下面定义了 fcntl 的命令。注意目前锁定命令还没有支持，而其它
   * 命令实际上还没有测试过。
   */
// 文件句柄(描述符)操作函数 fcntl() 的命令。
23 #define F_DUPFD      0      /* dup */ // 拷贝文件句柄为最小数值的句柄。
24 #define F_GETFD      1      /* get f_flags */ // 取文件句柄标志。
25 #define F_SETFD      2      /* set f_flags */ // 设置文件句柄标志。
26 #define F_GETFL      3      /* more flags (cloexec) */ // 取文件状态标志和访问模式。
27 #define F_SETFL      4      // 设置文件状态标志和访问模式。
// 下面是文件锁定命令。fcntl() 的第三个参数 lock 是指向 flock 结构的指针。
28 #define F_GETLK      5      /* not implemented */ // 返回阻止锁定的 flock 结构。
29 #define F_SETLK      6      // 设置(F_RDLCK 或 F_WRLCK)或清除(F_UNLCK)锁定。
30 #define F_SETLKW     7      // 等待设置或清除锁定。
31
32 /* for F_[GET/SET]FL */
/* 用于 F_GETFL 或 F_SETFL */
// 在执行 exec() 簇函数时关闭文件句柄。(执行时关闭 - Close On EXECution)
33 #define FD_CLOEXEC    1      /* actually anything with low bit set goes */
/* 实际上只要低位为 1 即可 */
34
35 /* Ok, these are locking features, and aren't implemented at any
36  * level. POSIX wants them.
37  */
/* OK, 以下是锁定类型，任何函数中都还没有实现。POSIX 标准要求这些类型。
   */
38 #define F_RDLCK      0      // 共享或读文件锁定。
39 #define F_WRLCK      1      // 独占或写文件锁定。
40 #define F_UNLCK      2      // 文件解锁。
41
42 /* Once again - not implemented, but ... */
/* 同样 - 也还没有实现，但是... */
// 文件锁定操作数据结构。描述了受影响文件段的类型(l_type)、开始偏移(l_whence)、
// 相对偏移(l_start)、锁定长度(l_len)和实施锁定的进程 id。
43 struct flock {
44     short l_type;          // 锁定类型 (F_RDLCK, F_WRLCK, F_UNLCK)。
45     short l_whence;        // 开始偏移 (SEEK_SET, SEEK_CUR 或 SEEK_END)。
46     off_t l_start;         // 阻塞锁定的开始处。相对偏移 (字节数)。
47     off_t l_len;           // 阻塞锁定的大小；如果是 0 则为到文件末尾。
48     pid_t l_pid;           // 加锁的进程 id。
49 };
50
// 以下是使用上述标志或命令的函数原型。
// 创建新文件或重写一个已存在文件。
// 参数 filename 是欲创建文件的文件名，mode 是创建文件的属性 (参见 include/sys/stat.h)。
51 extern int creat(const char * filename, mode_t mode);
// 文件句柄操作，会影响文件的打开。
// 参数 fildes 是文件句柄，cmd 是操作命令，见上面 23-30 行。
52 extern int fcntl(int fildes, int cmd, ...);
// 打开文件。在文件与文件句柄之间建立联系。
// 参数 filename 是欲打开文件的文件名，flags 是上面 7-17 行上的标志的组合。
53 extern int open(const char * filename, int flags, ...);

```


54
55 #endif
56

11.8 signal.h 文件

11.8.1 功能描述

11.8.2 文件注释

列表 linux/include/signal.h 文件

```

1 #ifndef \_SIGNAL\_H
2 #define \_SIGNAL\_H
3
4 #include <sys/types.h> // 类型头文件。定义了基本的系统数据类型。
5
6 typedef int sig\_atomic\_t; // 定义信号原子操作类型。
7 typedef unsigned int sigset\_t; /* 32 bits */ // 定义信号集类型。
8
9 #define \_NSIG 32 // 定义信号种类 -- 32 种。
10 #define NSIG \_NSIG // NSIG = _NSIG
11
12 // 以下这些是 Linux 0.11 内核中定义的信号。
13 #define SIGHUP 1 // Hang Up -- 挂断控制终端或进程。
14 #define SIGINT 2 // Interrupt -- 来自键盘的中断。
15 #define SIGQUIT 3 // Quit -- 来自键盘的退出。
16 #define SIGILL 4 // Illeagle -- 非法指令。
17 #define SIGTRAP 5 // Trap -- 跟踪断点。
18 #define SIGABRT 6 // Abort -- 异常结束。
19 #define SIGIOT 6 // IO Trap -- 同上。
20 #define SIGUNUSED 7 // Unused -- 没有使用。
21 #define SIGFPE 8 // FPE -- 协处理器出错。
22 #define SIGKILL 9 // Kill -- 强迫进程终止。
23 #define SIGUSR1 10 // User1 -- 用户信号 1，进程可使用。
24 #define SIGSEGV 11 // Segment Violation -- 无效内存引用。
25 #define SIGUSR2 12 // User2 -- 用户信号 2，进程可使用。
26 #define SIGPIPE 13 // Pipe -- 管道写出错，无读者。
27 #define SIGALRM 14 // Alarm -- 实时定时器报警。
28 #define SIGTERM 15 // Terminate -- 进程终止。
29 #define SIGSTKFLT 16 // Stack Fault -- 栈出错（协处理器）。
30 #define SIGCHLD 17 // Child -- 子进程停止或被终止。
31 #define SIGCONT 18 // Continue -- 恢复进程继续执行。
32 #define SIGSTOP 19 // Stop -- 停止进程的执行。
33 #define SIGTSTP 20 // TTY Stop -- tty 发出停止进程，可忽略。
34 #define SIGTTIN 21 // TTY In -- 后台进程请求输入。
35 #define SIGTTOU 22 // TTY Out -- 后台进程请求输出。
36
37 /* Ok, I haven't implemented sigactions, but trying to keep headers POSIX */

```

```

/* OK, 我还没有实现 sigactions 的编制, 但在头文件中仍希望遵守 POSIX 标准 */
37 #define SA_NOCLDSTOP 1 // 当子进程处于停止状态, 就不对 SIGCHLD 处理。
38 #define SA_NOMASK 0x40000000 // 不阻止在指定的信号处理程序(信号句柄)中再收到该信号。
39 #define SA_ONESHOT 0x80000000 // 信号句柄一旦被调用过就恢复到默认处理句柄。
40
// 以下参数用于 sigprocmask() -- 改变阻塞信号集(屏蔽码)。这些参数可以改变该函数的行为。
41 #define SIG_BLOCK 0 /* for blocking signals */
// 在阻塞信号集中加上给定的信号集。
42 #define SIG_UNBLOCK 1 /* for unblocking signals */
// 从阻塞信号集中删除指定的信号集。
43 #define SIG_SETMASK 2 /* for setting the signal mask */
// 设置阻塞信号集(信号屏蔽码)。

44
45 #define SIG_DFL ((void (*)(int))0) /* default signal handling */
// 默认的信号处理程序(信号句柄)。
46 #define SIG_IGN ((void (*)(int))1) /* ignore signal */
// 忽略信号的处理程序。

47
// 下面是 sigaction 的数据结构。
// sa_handler 是对应某信号指定要采取的行动。可以是上面的 SIG_DFL, 或者是 SIG_IGN 来忽略
// 该信号, 也可以是指向处理该信号函数的一个指针。
// sa_mask 给出了对信号的屏蔽码, 在信号程序执行时将阻塞对这些信号的处理。
// sa_flags 指定改变信号处理过程的信号集。它是由 37-39 行的位标志定义的。
// sa_restorer 恢复过程指针, 是用于保存原返回的过程指针。
// 另外, 引起触发信号处理的信号也将被阻塞, 除非使用了 SA_NOMASK 标志。
48 struct sigaction {
49     void (*sa_handler)(int);
50     sigset_t sa_mask;
51     int sa_flags;
52     void (*sa_restorer)(void);
53 };
54
// 为信号_sig 安装一个新的信号处理程序(信号句柄), 与 sigaction() 类似。
55 void (*signal(int _sig, void (*_func)(int)))(int);
// 向当前进程发送一个信号。其作用等价于 kill(getpid(), sig)。
56 int raise(int sig);
// 可用于向任何进程组或进程发送任何信号。
57 int kill(pid_t pid, int sig);
// 向信号集中添加信号。
58 int sigaddset(sigset_t *mask, int signo);
// 从信号集中去除指定的信号。
59 int sigdelset(sigset_t *mask, int signo);
// 从信号集中清除指定信号集。
60 int sigemptyset(sigset_t *mask);
// 向信号集中置入所有信号。
61 int sigfillset(sigset_t *mask);
// 判断一个信号是否是信号集中的。1 -- 是, 0 -- 不是, -1 -- 出错。
62 int sigismember(sigset_t *mask, int signo); /* 1 - is, 0 - not, -1 error */
// 对 set 中的信号进行检测, 看是否有挂起的信号。
63 int sigpending(sigset_t *set);
// 改变目前的被阻塞信号集(信号屏蔽码)。
64 int sigprocmask(int how, sigset_t *set, sigset_t *oldset);
// 用 sigmask 临时替换进程的信号屏蔽码, 然后暂停该进程直到收到一个信号。

```

```

65 int sigsuspend(sigset_t *sigmask);
    // 用于改变进程在收到指定信号时所采取的行动。
66 int sigaction(int sig, struct sigaction *act, struct sigaction *oldact);
67
68 #endif /* _SIGNAL_H */
69

```

11.9 stdarg.h 文件

11.9.1 功能描述

stdarg.h 是标准参数头文件。它以宏的形式定义变量参数列表。主要说明了一个类型(va_list)和三个宏(va_start, va_arg 和 va_end), 用于 vsprintf、vprintf、vfprintf 函数。

11.9.2 代码注释

列表 linux/include/stdarg.h 文件

```

1 #ifndef _STDARG_H
2 #define _STDARG_H
3
4 typedef char *va_list; // 定义 va_list 是一个字符指针类型。
5
6 /* Amount of space required in an argument list for an arg of type TYPE.
7    TYPE may alternatively be an expression whose type is used. */
8 /* 下面给出了类型为 TYPE 的 arg 参数列表所要求的空间容量。
9    TYPE 也可以是使用该类型的一个表达式 */
10
11 // 下面这句定义了取整后的 TYPE 类型的字节长度值。是 int 长度(4)的倍数。
12 #define __va_rounded_size(TYPE) \
13     (((sizeof (TYPE) + sizeof (int) - 1) / sizeof (int)) * sizeof (int))
14
15 // 下面这个函数(用宏实现)使 AP 指向传给函数的可变参数表的第一个参数。
16 // 在第一次调用 va_arg 或 va_end 之前, 必须首先调用该函数。
17 #ifndef __sparc__
18 #define va_start(AP, LASTARG) \
19     (AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
20 #else
21 #define va_start(AP, LASTARG) \
22     (__builtin_saveregs (), \
23     AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
24 #endif
25
26 // 下面该宏用于被调用函数完成一次正常返回。va_end 可以修改 AP 使其在重新调用
27 // va_start 之前不能被使用。va_end 必须在 va_arg 读完所有的参数后再被调用。
28 void va_end (va_list); /* Defined in gnu lib */ /* 在 gnu lib 中定义 */
29 #define va_end(AP)
30
31 // 下面该宏用于扩展表达式使其与下一个被传递参数具有相同的类型和值。
32 // 对于缺省值, va_arg 可以用字符、无符号字符和浮点类型。
33 // 在第一次使用 va_arg 时, 它返回表中的第一个参数, 后续的每次调用都将返回表中的

```

```

// 下一个参数。这是通过先访问 AP，然后把它增加以指向下一项来实现的。
// va_arg 使用 TYPE 来完成访问和定位下一项，每调用一次 va_arg，它就修改 AP 以指示
// 表中的下一参数。
24 #define va_arg(AP, TYPE) \
25 (AP += __va_rounded_size (TYPE), \
26 *((TYPE *) (AP - __va_rounded_size (TYPE))))
27
28 #endif /* _STDARG_H */
29

```

11.10 stddef.h 文件

11.10.1 功能描述

该文件定义了一些常用的类型和宏。内核中很少使用该文件。

11.10.2 代码注释

列表 linux/include/stddef.h 文件

```

1 #ifndef _STDDEF_H
2 #define _STDDEF_H
3
4 #ifndef PTRDIFF_T
5 #define PTRDIFF_T
6 typedef long ptrdiff_t; // 两个指针相减结果的类型。
7 #endif
8
9 #ifndef SIZE_T
10 #define SIZE_T
11 typedef unsigned long size_t; // sizeof 返回的类型。
12 #endif
13
14 #undef NULL
15 #define NULL ((void *)0) // 空指针。
16
17 #define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER) // 成员在类型中的偏移位置。
18
19 #endif
20

```

11.11 string.h 文件

11.11.1 功能描述

该头文件中以内嵌函数的形式定义了所有字符串操作函数，为了提高执行速度使用了内嵌汇编程序。

11.11.2 代码注释

列表 linux/include/string.h 文件

```

1 #ifndef STRING\_H
2 #define STRING\_H
3
4 #ifndef NULL
5 #define NULL ((void *) 0)
6 #endif
7
8 #ifndef SIZE\_T
9 #define SIZE\_T
10 typedef unsigned int size\_t;
11 #endif
12
13 extern char * strerror(int errno);
14
15 /*
16  * This string-include defines all string functions as inline
17  * functions. Use gcc. It also assumes ds=es=data space, this should be
18  * normal. Most of the string-functions are rather heavily hand-optimized,
19  * see especially strtok, strstr, str[c]spn. They should work, but are not
20  * very easy to understand. Everything is done entirely within the register
21  * set, making the functions fast and clean. String instructions have been
22  * used through-out, making for "slightly" unclear code :-))
23  *
24  * (C) 1991 Linus Torvalds
25  */
26
27 /*
28  * 这个字符串头文件以内嵌函数的形式定义了所有字符串操作函数。使用 gcc 时，同时
29  * 假定了 ds=es=数据空间，这应该是常规的。绝大多数字符串函数都是经手工进行大量
30  * 优化的，尤其是函数 strtok、strstr、str[c]spn。它们应该能正常工作，但却不是那
31  * 么容易理解。所有的操作基本上都是使用寄存器集来完成的，这使得函数即快有整洁。
32  * 所有地方都使用了字符串指令，这又使得代码“稍微”难以理解☺
33  *
34  * (C) 1991 Linus Torvalds
35  */
36
37 // 将一个字符串(src)拷贝到另一个字符串(dest)，直到遇到 NULL 字符后停止。
38 // 参数: dest - 目的字符串指针, src - 源字符串指针。
39 // %0 - esi(src), %1 - edi(dest)。
40 extern inline char * strcpy(char * dest, const char *src)
41 {
42     __asm__ ("cld\n" // 清方向位。
43             "l:|t lodsb\n|t" // 加载 DS:[esi]处 1 字节→al, 并更新 esi。
44             "stosb\n|t" // 存储字节 al→ES:[edi], 并更新 edi。
45             "testb %%al, %%al\n|t" // 刚存储的字节是 0?
46             "jne 1b" // 不是则向后跳转到标号 1 处, 否则结束。
47             :: "S" (src), "D" (dest): "si", "di", "ax");
48     return dest; // 返回目的字符串指针。
49 }
50
51 // 拷贝源字符串 count 个字节到目的字符串。

```

```

// 如果源串长度小于 count 个字节，就附加空字符(NULL)到目的字符串。
// 参数: dest - 目的字符串指针, src - 源字符串指针, count - 拷贝字节数。
// %0 - esi(src), %1 - edi(dest), %2 - ecx(count)。
38 extern inline char * strncpy(char * dest, const char *src, int count)
39 {
40     __asm__ ("cld\n"                                // 清方向位。
41             "l:|tdecl %2\n|t"                        // 寄存器 ecx-- (count--)。
42             "js 2f\n|t"                                // 如果 count<0 则向前跳转到标号 2，结束。
43             "lodsbl\n|t"                            // 取 ds:[esi]处 1 字节→al，并且 esi++。
44             "stosbl\n|t"                            // 存储该字节→es:[edi]，并且 edi++。
45             "testb %%al, %%al\n|t"                  // 该字节是 0?
46             "jne 1b\n|t"                            // 不是，则向前跳转到标号 1 处继续拷贝。
47             "rep\n|t"                                // 否则，在目的串中存放剩余个数的空字符。
48             "stosbl\n"
49             "2:"
50             :: "S" (src), "D" (dest), "c" (count): "si", "di", "ax", "cx");
51 return dest;                                // 返回目的字符串指针。
52 }
53
//// 将源字符串拷贝到目的字符串的末尾处。
// 参数: dest - 目的字符串指针, src - 源字符串指针。
// %0 - esi(src), %1 - edi(dest), %2 - eax(0), %3 - ecx(-1)。
54 extern inline char * strcat(char * dest, const char * src)
55 {
56     __asm__ ("cld\n|t"                                // 清方向位。
57             "repne\n|t"                            // 比较 al 与 es:[edi]字节，并更新 edi++，
58             "scasbl\n|t"                            // 直到找到目的串中是 0 的字节，此时 edi 已经指向后 1 字
59             "decl %1\n|t"                            // 节。
60             "l:|tlodsb\n|t"                        // 让 es:[edi]指向 0 值字节。
61             "stosbl\n|t"                            // 取源字符串字节 ds:[esi]→al，并 esi++。
62             "testb %%al, %%al\n|t"                  // 将该字节存到 es:[edi]，并 edi++。
63             "jne 1b"                                // 该字节是 0?
64             :: "S" (src), "D" (dest), "a" (0), "c" (0xffffffff): "si", "di", "ax", "cx");
65 return dest;                                // 不是，则向后跳转到标号 1 处继续拷贝，否则结束。
66 }
67
//// 将源字符串的 count 个字节复制到目的字符串的末尾处，最后添一空字符。
// 参数: dest - 目的字符串, src - 源字符串, count - 欲复制的字节数。
// %0 - esi(src), %1 - edi(dest), %2 - eax(0), %3 - ecx(-1), %4 - (count)。
68 extern inline char * strncat(char * dest, const char * src, int count)
69 {
70     __asm__ ("cld\n|t"                                // 清方向位。
71             "repne\n|t"                            // 比较 al 与 es:[edi]字节，edi++。
72             "scasbl\n|t"                            // 直到找到目的串的末端 0 值字节。
73             "decl %1\n|t"                            // edi 指向该 0 值字节。
74             "movl %4, %3\n|t"                        // 欲复制字节数→ecx。
75             "l:|tdecl %3\n|t"                        // ecx-- (从 0 开始计数)。
76             "js 2f\n|t"                                // ecx < 0 ?，是则向前跳转到标号 2 处。
77             "lodsbl\n|t"                            // 否则取 ds:[esi]处的字节→al，esi++。
78             "stosbl\n|t"                            // 存储到 es:[edi]处，edi++。
79             "testb %%al, %%al\n|t"                  // 该字节值为 0?
80             "jne 1b\n|t"                            // 不是则向后跳转到标号 1 处，继续复制。

```

```

81      "2:\txorl %2,%2\n\t"          // 将 al 清零。
82      "stosb"                        // 存到 es:[edi]处。
83      : "S" (src), "D" (dest), "a" (0), "c" (0xffffffff), "g" (count)
84      : "si", "di", "ax", "cx");
85  return dest;                      // 返回目的字符串指针。
86  }
87
88  /// 将一个字符串与另一个字符串进行比较。
89  /// 参数: cs - 字符串 1, ct - 字符串 2。
90  /// %0 - eax(__res)返回值, %1 - edi(cs)字符串 1 指针, %2 - esi(ct)字符串 2 指针。
91  /// 返回: 如果串 1 > 串 2, 则返回 1; 串 1 = 串 2, 则返回 0; 串 1 < 串 2, 则返回-1。
92  extern inline int strcmp(const char * cs, const char * ct)
93  {
94      register int __res __asm__( "ax"); // __res 是寄存器变量(eax)。
95      __asm__( "cld\n"                  // 清方向位。
96              "1:\tlodsb\n\t"          // 取字符串 2 的字节 ds:[esi]→al, 并且 esi++。
97              "scasb\n\t"              // al 与字符串 1 的字节 es:[edi]作比较, 并且 edi++。
98              "jne 2f\n\t"              // 如果不相等, 则向前跳转到标号 2。
99              "testb %%al, %%al\n\t"    // 该字节是 0 值字节吗(字符串结尾)?
100             "jne 1b\n\t"              // 不是, 则向后跳转到标号 1, 继续比较。
101             "xorl %%eax, %%eax\n\t"    // 是, 则返回值 eax 清零,
102             "jmp 3f\n\t"              // 向前跳转到标号 3, 结束。
103             "2:\tmovl $1, %%eax\n\t"   // eax 中置 1。
104             "jl 3f\n\t"              // 若前面比较中串 2 字符<串 1 字符, 则返回正值, 结束。
105             "negl %%eax\n\t"          // 否则 eax = -eax, 返回负值, 结束。
106             "3:"
107             : "=a" (__res) : "D" (cs), "S" (ct) : "si", "di");
108  return __res;                      // 返回比较结果。
109 }
110
111  /// 字符串 1 与字符串 2 的前 count 个字符进行比较。
112  /// 参数: cs - 字符串 1, ct - 字符串 2, count - 比较的字符数。
113  /// %0 - eax(__res)返回值, %1 - edi(cs)串 1 指针, %2 - esi(ct)串 2 指针, %3 - ecx(count)。
114  /// 返回: 如果串 1 > 串 2, 则返回 1; 串 1 = 串 2, 则返回 0; 串 1 < 串 2, 则返回-1。
115  extern inline int strncmp(const char * cs, const char * ct, int count)
116  {
117      register int __res __asm__( "ax"); // __res 是寄存器变量(eax)。
118      __asm__( "cld\n"                  // 清方向位。
119              "1:\tdecl %3\n\t"          // count--。
120              "js 2f\n\t"              // 如果 count<0, 则向前跳转到标号 2。
121              "lodsb\n\t"              // 取串 2 的字符 ds:[esi]→al, 并且 esi++。
122              "scasb\n\t"              // 比较 al 与串 1 的字符 es:[edi], 并且 edi++。
123              "jne 3f\n\t"              // 如果不相等, 则向前跳转到标号 3。
124              "testb %%al, %%al\n\t"    // 该字符是 NULL 字符吗?
125              "jne 1b\n\t"              // 不是, 则向后跳转到标号 1, 继续比较。
126              "2:\txorl %%eax, %%eax\n\t" // 是 NULL 字符, 则 eax 清零(返回值)。
127              "jmp 4f\n\t"              // 向前跳转到标号 4, 结束。
128              "3:\tmovl $1, %%eax\n\t"   // eax 中置 1。
129              "jl 4f\n\t"              // 如果前面比较中串 2 字符<串 2 字符, 则返回 1, 结束。
130              "negl %%eax\n\t"          // 否则 eax = -eax, 返回负值, 结束。
131              "4:"
132              : "=a" (__res) : "D" (cs), "S" (ct), "c" (count) : "si", "di", "cx");
133  return __res;                      // 返回比较结果。

```



```

126 }
127
128 // 在字符串中寻找第一个匹配的字符。
129 // 参数: s - 字符串, c - 欲寻找的字符。
130 // %0 - eax(__res), %1 - esi(字符串指针 s), %2 - eax(字符 c)。
131 // 返回: 返回字符串中第一次出现匹配字符的指针。若没有找到匹配的字符, 则返回空指针。
132 extern inline char * strchr(const char * s, char c)
133 {
134     register char * __res __asm__( "ax" ); // __res 是寄存器变量(eax)。
135     __asm__( "cld\n\t" // 清方向位。
136             "movb %%al, %%ah\n\t" // 将欲比较字符移到 ah。
137             "1:\tlodsb\n\t" // 取字符串中字符 ds:[esi]→al, 并且 esi++。
138             "cmpb %%ah, %%al\n\t" // 字符串中字符 al 与指定字符 ah 相比较。
139             "je 2f\n\t" // 若相等, 则向前跳转到标号 2 处。
140             "testb %%al, %%al\n\t" // al 中字符是 NULL 字符吗? (字符串结尾?)
141             "jne 1b\n\t" // 若不是, 则向后跳转到标号 1, 继续比较。
142             "movl $1, %1\n\t" // 是, 则说明没有找到匹配字符, esi 置 1。
143             "2:\tmovl %1, %0\n\t" // 将指向匹配字符后一个字节处的指针值放入 eax
144             "decl %0" // 将指针调整为指向匹配的字符。
145             : "=a" (__res) : "S" (s), "" (c) : "si" );
146     return __res; // 返回指针。
147 }
148
149 // 寻找字符串中指定字符最后一次出现的地方。(反向搜索字符串)
150 // 参数: s - 字符串, c - 欲寻找的字符。
151 // %0 - edx(__res), %1 - edx(0), %2 - esi(字符串指针 s), %3 - eax(字符 c)。
152 // 返回: 返回字符串中最后一次出现匹配字符的指针。若没有找到匹配的字符, 则返回空指针。
153 extern inline char * strrchr(const char * s, char c)
154 {
155     register char * __res __asm__( "dx" ); // __res 是寄存器变量(edx)。
156     __asm__( "cld\n\t" // 清方向位。
157             "movb %%al, %%ah\n\t" // 将欲寻找的字符移到 ah。
158             "1:\tlodsb\n\t" // 取字符串中字符 ds:[esi]→al, 并且 esi++。
159             "cmpb %%ah, %%al\n\t" // 字符串中字符 al 与指定字符 ah 作比较。
160             "jne 2f\n\t" // 若不相等, 则向前跳转到标号 2 处。
161             "movl %%esi, %0\n\t" // 将字符指针保存到 edx 中。
162             "decl %0\n\t" // 指针后退一位, 指向字符串中匹配字符处。
163             "2:\tttestb %%al, %%al\n\t" // 比较的字符是 0 吗(到字符串尾)?
164             "jne 1b" // 不是则向后跳转到标号 1 处, 继续比较。
165             : "=d" (__res) : "" (0), "S" (s), "a" (c) : "ax", "si" );
166     return __res; // 返回指针。
167 }
168
169 // 在字符串 1 中寻找第 1 个字符序列, 该字符序列中的任何字符都包含在字符串 2 中。
170 // 参数: cs - 字符串 1 指针, ct - 字符串 2 指针。
171 // %0 - esi(__res), %1 - eax(0), %2 - ecx(-1), %3 - esi(串 1 指针 cs), %4 - (串 2 指针 ct)。
172 // 返回字符串 1 中包含字符串 2 中任何字符的首个字符序列的长度值。
173 extern inline int strspn(const char * cs, const char * ct)
174 {
175     register char * __res __asm__( "si" ); // __res 是寄存器变量(es:esi)。
176     __asm__( "cld\n\t" // 清方向位。
177             "movl %4, %%edi\n\t" // 首先计算串 2 的长度。串 2 指针放入 edi 中。
178             "repne\n\t" // 比较 al(0) 与串 2 中的字符(es:[edi]), 并 edi++。

```



```

167     "scasb|n|t" // 如果不相等就继续比较(ecx 逐步递减)。
168     "notl %%ecx|n|t" // ecx 中每位取反。
169     "decl %%ecx|n|t" // ecx--, 得串 2 的长度值。
170     "movl %%ecx, %%edx|n" // 将串 2 的长度值暂放入 edx 中。
171     "l:|tlodsb|n|t" // 取串 1 字符 ds:[esi]→al, 并且 esi++。
172     "testb %%al, %%al|n|t" // 该字符等于 0 值吗(串 1 结尾)?
173     "je 2f|n|t" // 如果是, 则向前跳转到标号 2 处。
174     "movl %4, %%edi|n|t" // 取串 2 头指针放入 edi 中。
175     "movl %%edx, %%ecx|n|t" // 再将串 2 的长度值放入 ecx 中。
176     "repne|n|t" // 比较 al 与串 2 中字符 es:[edi], 并且 edi++。
177     "scasb|n|t" // 如果不相等就继续比较。
178     "je 1b|n" // 如果相等, 则向后跳转到标号 1 处。
179     "2:|tdecl %0" // esi--, 指向最后一个包含在串 2 中的字符。
180     : "=S" (__res): "a" (0), "c" (0xffffffff), "" (cs), "g" (ct)
181     : "ax", "cx", "dx", "di");
182 return __res-cs; // 返回字符序列的长度值。
183 }
184
185 // 寻找字符串 1 中不包含字符串 2 中任何字符的首个字符序列。
186 // 参数: cs - 字符串 1 指针, ct - 字符串 2 指针。
187 // %0 - esi(__res), %1 - eax(0), %2 - ecx(-1), %3 - esi(串 1 指针 cs), %4 - (串 2 指针 ct)。
188 // 返回字符串 1 中不包含字符串 2 中任何字符的首个字符序列的长度值。
189 extern inline int strcspn(const char * cs, const char * ct)
190 {
191     register char * __res __asm__ ("si"); // __res 是寄存器变量(esi)。
192     __asm__ ("cld|n|t" // 清方向位。
193             "movl %4, %%edi|n|t" // 首先计算串 2 的长度。串 2 指针放入 edi 中。
194             "repne|n|t" // 比较 al(0)与串 2 中的字符(es:[edi]), 并 edi++。
195             "scasb|n|t" // 如果不相等就继续比较(ecx 逐步递减)。
196             "notl %%ecx|n|t" // ecx 中每位取反。
197             "decl %%ecx|n|t" // ecx--, 得串 2 的长度值。
198             "movl %%ecx, %%edx|n" // 将串 2 的长度值暂放入 edx 中。
199             "l:|tlodsb|n|t" // 取串 1 字符 ds:[esi]→al, 并且 esi++。
200             "testb %%al, %%al|n|t" // 该字符等于 0 值吗(串 1 结尾)?
201             "je 2f|n|t" // 如果是, 则向前跳转到标号 2 处。
202             "movl %4, %%edi|n|t" // 取串 2 头指针放入 edi 中。
203             "movl %%edx, %%ecx|n|t" // 再将串 2 的长度值放入 ecx 中。
204             "repne|n|t" // 比较 al 与串 2 中字符 es:[edi], 并且 edi++。
205             "scasb|n|t" // 如果不相等就继续比较。
206             "jne 1b|n" // 如果不相等, 则向后跳转到标号 1 处。
207             "2:|tdecl %0" // esi--, 指向最后一个包含在串 2 中的字符。
208             : "=S" (__res): "a" (0), "c" (0xffffffff), "" (cs), "g" (ct)
209             : "ax", "cx", "dx", "di");
210 return __res-cs; // 返回字符序列的长度值。
211 }
212
213 // 在字符串 1 中寻找首个包含在字符串 2 中的任何字符。
214 // 参数: cs - 字符串 1 的指针, ct - 字符串 2 的指针。
215 // %0 -esi(__res), %1 -eax(0), %2 -ecx(0xffffffff), %3 -esi(串 1 指针 cs), %4 -(串 2 指针 ct)。
216 // 返回字符串 1 中首个包含字符串 2 中字符的指针。
217 extern inline char * strpbrk(const char * cs, const char * ct)
218 {
219     register char * __res __asm__ ("si"); // __res 是寄存器变量(esi)。

```

```

212 __asm__( "cld\n\t" // 清方向位。
213 "movl %4,%%edi\n\t" // 首先计算串 2 的长度。串 2 指针放入 edi 中。
214 "repne\n\t" // 比较 al(0) 与串 2 中的字符(es:[edi])，并 edi++。
215 "scasb\n\t" // 如果不相等就继续比较(ecx 逐步递减)。
216 "notl %%ecx\n\t" // ecx 中每位取反。
217 "decl %%ecx\n\t" // ecx--，得串 2 的长度值。
218 "movl %%ecx,%%edx\n\t" // 将串 2 的长度值暂放入 edx 中。
219 "l:|t|lodsbl\n\t" // 取串 1 字符 ds:[esi]→al，并且 esi++。
220 "testb %al,%al\n\t" // 该字符等于 0 值吗(串 1 结尾)?
221 "je 2f\n\t" // 如果是，则向前跳转到标号 2 处。
222 "movl %4,%%edi\n\t" // 取串 2 头指针放入 edi 中。
223 "movl %%edx,%%ecx\n\t" // 再将串 2 的长度值放入 ecx 中。
224 "repne\n\t" // 比较 al 与串 2 中字符 es:[edi]，并且 edi++。
225 "scasb\n\t" // 如果不相等就继续比较。
226 "jne 1b\n\t" // 如果不相等，则向后跳转到标号 1 处。
227 "decl %0\n\t" // esi--，指向一个包含在串 2 中的字符。
228 "jmp 3f\n\t" // 向前跳转到标号 3 处。
229 "2:|txorl %0,%0\n\t" // 没有找到符合条件的，将返回值为 NULL。
230 "3:"
231 : "=S" (__res): "a" (0), "c" (0xffffffff), "" (cs), "g" (ct)
232 : "ax", "cx", "dx", "di");
233 return __res; // 返回指针值。
234 }
235
236 // 在字符串 1 中寻找首个匹配整个字符串 2 的字符串。
237 // 参数: cs - 字符串 1 的指针, ct - 字符串 2 的指针。
238 // %0 -eax(__res), %1 -eax(0), %2 -ecx(0xffffffff), %3 -esi(串 1 指针 cs), %4 -(串 2 指针 ct)。
239 // 返回: 返回字符串 1 中首个匹配字符串 2 的字符串指针。
240 extern inline char * strstr(const char * cs,const char * ct)
241 {
242 register char * __res __asm__( "ax"); // __res 是寄存器变量(eax)。
243 __asm__( "cld\n\t" \
244 "movl %4,%%edi\n\t" // 清方向位。
245 "repne\n\t" // 首先计算串 2 的长度。串 2 指针放入 edi 中。
246 "scasb\n\t" // 比较 al(0) 与串 2 中的字符(es:[edi])，并 edi++。
247 "notl %%ecx\n\t" // 如果不相等就继续比较(ecx 逐步递减)。
248 "decl %%ecx\n\t" // ecx 中每位取反。
249 /* NOTE! This also sets Z if searchstring='' */
250 /* 注意! 如果搜索串为空, 将设置 Z 标志 */ // 得串 2 的长度值。
251 "movl %%ecx,%%edx\n\t" // 将串 2 的长度值暂放入 edx 中。
252 "l:|tmovl %4,%%edi\n\t" // 取串 2 头指针放入 edi 中。
253 "movl %%esi,%%eax\n\t" // 将串 1 的指针复制到 eax 中。
254 "movl %%edx,%%ecx\n\t" // 再将串 2 的长度值放入 ecx 中。
255 "repe\n\t" // 比较串 1 和串 2 字符(ds:[esi], es:[edi]), esi++,edi++。
256 "cmpsb\n\t" // 若对应字符相等就一直比较下去。
257 "je 2f\n\t" // 若全相等, 则转到标号 2。
258 "xchgl %%eax,%%esi\n\t" // 串 1 头指针→esi, 比较结果的串 1 指针→eax。
259 "incl %%esi\n\t" // 串 1 头指针指向下一个字符。
260 "cmpb $0,-1(%%eax)\n\t" // 串 1 指针(eax-1)所指字节是 0 吗?
261 "jne 1b\n\t" // 不是则跳转到标号 1, 继续从串 1 的第 2 个字符开始比较。
262 "xorl %%eax,%%eax\n\t" // 清 eax, 表示没有找到匹配。
263 "2:"
264 : "=a" ( __res): "" (0), "c" (0xffffffff), "S" (cs), "g" (ct)

```

```

259         : "cx", "dx", "di", "si");
260 return __res;                // 返回比较结果。
261 }
262
263 // 计算字符串长度。
264 // 参数: s - 字符串。
265 // %0 - ecx(__res), %1 - edi(字符串指针 s), %2 - eax(0), %3 - ecx(0xffffffff)。
266 // 返回: 返回字符串的长度。
267 extern inline int strlen(const char * s)
268 {
269     register int __res __asm__( "cx");    // __res 是寄存器变量(ecx)。
270     __asm__( "cld\n\t"                    // 清方向位。
271             "repne\n\t"                  // al(0)与字符串中字符 es:[edi]比较,
272             "scasb\n\t"                  // 若不相等就一直比较。
273             "notl %0\n\t"                // ecx 取反。
274             "decl %0"                    // ecx--, 得字符串得长度值。
275             : "=c" (__res) : "D" (s), "a" (0), "" (0xffffffff) : "di");
276 return __res;                    // 返回字符串长度值。
277 }
278
279 extern char * __strtok;        // 用于临时存放指向下面被分析字符串 1(s) 的指针。
280
281 // 利用字符串 2 中的字符将字符串 1 分割成标记(token)序列。
282 // 将串 1 看作是包含零个或多个单词(token)的序列, 并由分割符字符串 2 中的一个或多个字符分开。
283 // 第一次调用 strtok() 时, 将返回指向字符串 1 中第 1 个 token 首字符的指针, 并在返回 token 时将
284 // 一 null 字符写到分割符处。后续使用 null 作为字符串 1 的调用, 将用这种方法继续扫描字符串 1,
285 // 直到没有 token 为止。在不同的调用过程中, 分割符串 2 可以不同。
286 // 参数: s - 待处理的字符串 1, ct - 包含各个分割符的字符串 2。
287 // 汇编输出: %0 - ebx(__res), %1 - esi(__strtok);
288 // 汇编输入: %2 - ebx(__strtok), %3 - esi(字符串 1 指针 s), %4 - (字符串 2 指针 ct)。
289 // 返回: 返回字符串 s 中第 1 个 token, 如果没有找到 token, 则返回一个 null 指针。
290 // 后续使用字符串 s 指针为 null 的调用, 将在原字符串 s 中搜索下一个 token。
291 extern inline char * strtok(char * s, const char * ct)
292 {
293     register char * __res __asm__( "si");
294     __asm__( "testl %1, %1\n\t"          // 首先测试 esi(字符串 1 指针 s) 是否是 NULL。
295             "jne 1f\n\t"                // 如果不是, 则表明是首次调用本函数, 跳转标号 1。
296             "testl %0, %0\n\t"          // 如果是 NULL, 则表示此次是后续调用, 测 ebx(__strtok)。
297             "je 8f\n\t"                 // 如果 ebx 指针是 NULL, 则不能处理, 跳转结束。
298             "movl %0, %1\n\t"           // 将 ebx 指针复制到 esi。
299             "l: |txorl %0, %0\n\t"       // 清 ebx 指针。
300             "movl $-1, %%ecx\n\t"        // 置 ecx = 0xffffffff。
301             "xorl %%eax, %%eax\n\t"      // 清零 eax。
302             "cld\n\t"                   // 清方向位。
303             "movl %4, %%edi\n\t"         // 下面求字符串 2 的长度。edi 指向字符串 2。
304             "repne\n\t"                  // 将 al(0)与 es:[edi]比较, 并且 edi++。
305             "scasb\n\t"                  // 直到找到字符串 2 的结束 null 字符, 或计数 ecx==0。
306             "notl %%ecx\n\t"             // 将 ecx 取反,
307             "decl %%ecx\n\t"             // ecx--, 得到字符串 2 的长度值。
308             "je 7f\n\t"                  /* empty delimiter-string */
309                                         /* 分割符字符串空 */ // 若串 2 长度为 0, 则转标号 7。
310             "movl %%ecx, %%edx\n\t"      // 将串 2 长度暂存入 edx。
311             "2: |t lodsb\n\t"            // 取串 1 的字符 ds:[esi]→al, 并且 esi++。

```

```

297     "testb %%al, %%al\n\t"           // 该字符为 0 值吗(串 1 结束)?
298     "je 7f\n\t"                     // 如果是, 则跳转标号 7。
299     "movl %4, %%edi\n\t"            // edi 再次指向串 2 首。
300     "movl %%edx, %%ecx\n\t"          // 取串 2 的长度值置入计数器 ecx。
301     "repne\n\t"                      // 将 a1 中串 1 的字符与串 2 中所有字符比较,
302     "scasb\n\t"                     // 判断该字符是否为分割符。
303     "je 2b\n\t"                      // 若能在串 2 中找到相同字符(分割符), 则跳转标号 2。
304     "decl %1\n\t"                   // 若不是分割符, 则串 1 指针 esi 指向此时的该字符。
305     "cmpb $0, (%1)\n\t"              // 该字符是 NULL 字符吗?
306     "je 7f\n\t"                     // 若是, 则跳转标号 7 处。
307     "movl %1, %0\n\t"                // 将该字符的指针 esi 存放在 ebx。
308     "3:\t\tlodsb\n\t"                // 取串 1 下一个字符 ds:[esi]→al, 并且 esi++。
309     "testb %%al, %%al\n\t"           // 该字符是 NULL 字符吗?
310     "je 5f\n\t"                      // 若是, 表示串 1 结束, 跳转到标号 5。
311     "movl %4, %%edi\n\t"            // edi 再次指向串 2 首。
312     "movl %%edx, %%ecx\n\t"          // 串 2 长度值置入计数器 ecx。
313     "repne\n\t"                      // 将 a1 中串 1 的字符与串 2 中每个字符比较,
314     "scasb\n\t"                     // 测试 a1 字符是否是分割符。
315     "jne 3b\n\t"                    // 若不是分割符则跳转标号 3, 检测串 1 中下一个字符。
316     "decl %1\n\t"                   // 若是分割符, 则 esi--, 指向该分割符字符。
317     "cmpb $0, (%1)\n\t"              // 该分割符是 NULL 字符吗?
318     "je 5f\n\t"                     // 若是, 则跳转到标号 5。
319     "movb $0, (%1)\n\t"              // 若不是, 则将该分割符用 NULL 字符替换掉。
320     "incl %1\n\t"                   // esi 指向串 1 中下一个字符, 也即剩余串首。
321     "jmp 6f\n\t"                    // 跳转标号 6 处。
322     "5:\txorl %1, %1\n\t"            // esi 清零。
323     "6:\t\tcmpb $0, (%0)\n\t"          // ebx 指针指向 NULL 字符吗?
324     "jne 7f\n\t"                    // 若不是, 则跳转标号 7。
325     "xorl %0, %0\n\t"                // 若是, 则让 ebx=NULL。
326     "7:\t\ttestl %0, %0\n\t"           // ebx 指针为 NULL 吗?
327     "jne 8f\n\t"                    // 若不是则跳转 8, 结束汇编代码。
328     "movl %0, %1\n\t"                // 将 esi 置为 NULL。
329     "8:"
330     : "=b" (__res), "=S" (__strtok)
331     : "" (__strtok), "I" (s), "g" (ct)
332     : "ax", "cx", "dx", "di");
333 return __res;                       // 返回指向新 token 的指针。
334 }
335
336 // 内存块复制。从源地址 src 处开始复制 n 个字节到目的地址 dest 处。
337 // 参数: dest - 复制的目的地址, src - 复制的源地址, n - 复制字节数。
338 // %0 - ecx(n), %1 - esi(src), %2 - edi(dest)。
339 extern inline void * memcpy(void * dest, const void * src, int n)
340 {
341     __asm__( "cld\n\t"                // 清方向位。
342             "rep\n\t"                // 重复执行复制 ecx 个字节,
343             "movsb"                  // 从 ds:[esi]到 es:[edi], esi++, edi++。
344             : : "c" (n), "S" (src), "D" (dest)
345             : "cx", "si", "di");
346 return dest;                         // 返回目的地址。
347 }
348
349 // 内存块移动。同内存块复制, 但考虑移动的方向。

```

```

// 参数: dest - 复制的目的地址, src - 复制的源地址, n - 复制字节数。
// 若 dest<src 则: %0 - ecx(n), %1 - esi(src), %2 - edi(dest)。
// 否则: %0 - ecx(n), %1 - esi(src+n-1), %2 - edi(dest+n-1)。
// 这样操作是为了防止在复制时错误地重叠覆盖。
346 extern inline void * memmove(void * dest, const void * src, int n)
347 {
348     if (dest<src)
349         __asm__( "cld\n\t"           // 清方向位。
350                 "rep\n\t"           // 从 ds:[esi]到 es:[edi], 并且 esi++, edi++,
351                 "movsb"             // 重复执行复制 ecx 字节。
352                 :: "c" (n), "S" (src), "D" (dest)
353                 : "cx", "si", "di");
354     else
355         __asm__( "std\n\t"           // 置方向位, 从末端开始复制。
356                 "rep\n\t"           // 从 ds:[esi]到 es:[edi], 并且 esi--, edi--,
357                 "movsb"             // 复制 ecx 个字节。
358                 :: "c" (n), "S" (src+n-1), "D" (dest+n-1)
359                 : "cx", "si", "di");
360     return dest;
361 }
362
//// 比较 n 个字节的内存块(两个字符串), 即使遇上 NULL 字节也不停止比较。
// 参数: cs - 内存块 1 地址, ct - 内存块 2 地址, count - 比较的字节数。
// %0 - eax(__res), %1 - eax(0), %2 - edi(内存块 1), %3 - esi(内存块 2), %4 - ecx(count)。
// 返回: 若块 1>块 2 返回 1; 块 1<块 2, 返回-1; 块 1==块 2, 则返回 0。
363 extern inline int memcmp(const void * cs, const void * ct, int count)
364 {
365     register int __res __asm__("ax"); // __res 是寄存器变量。
366     __asm__( "cld\n\t"           // 清方向位。
367             "repe\n\t"          // 如果相等则重复,
368             "cmprsb\n\t"        // 比较 ds:[esi]与 es:[edi]的内容, 并且 esi++, edi++。
369             "je 1f\n\t"         // 如果都相同, 则跳转到标号 1, 返回 0(eax)值
370             "movl $1, %%eax\n\t" // 否则 eax 置 1,
371             "jl 1f\n\t"         // 若内存块 2 内容的值<内存块 1, 则跳转标号 1。
372             "negl %%eax\n\t"     // 否则 eax = -eax。
373             "1:"
374             : "=a" (__res): "" (0), "D" (cs), "S" (ct), "c" (count)
375             : "si", "di", "cx");
376     return __res; // 返回比较结果。
377 }
378
//// 在 n 字节大小的内存块(字符串)中寻找指定字符。
// 参数: cs - 指定内存块地址, c - 指定的字符, count - 内存块长度。
// %0 - edi(__res), %1 - eax(字符 c), %2 - edi(内存块地址 cs), %3 - ecx(字节数 count)。
// 返回第一个匹配字符的指针, 如果没有找到, 则返回 NULL 字符。
379 extern inline void * memchr(const void * cs, char c, int count)
380 {
381     register void * __res __asm__("di"); // __res 是寄存器变量。
382     if (!count) // 如果内存块长度==0, 则返回 NULL, 没有找到。
383         return NULL;
384     __asm__( "cld\n\t"           // 清方向位。
385             "repne\n\t"         // 如果不相等则重复执行下面语句,
386             "scasb\n\t"         // al 中字符与 es:[edi]字符作比较, 并且 edi++,

```

```

387     "je lf\n\t"           // 如果相等则向前跳转到标号 1 处。
388     "movl $1,%0\n"       // 否则 edi 中置 1。
389     "l:\tdecl %0"         // 让 edi 指向找到的字符 (或是 NULL)。
390     : "=D" (__res): "a" (c), "D" (cs), "c" (count)
391     : "cx");
392 return __res;             // 返回字符指针。
393 }
394
395 // 用字符 c 填充指定长度内存块。
396 // 用字符 c 填充 s 指向的内存区域, 共填 count 字节。
397 // %0 - eax(字符 c), %1 - edi(内存地址), %2 - ecx(字节数 count)。
398 extern inline void * memset(void * s, char c, int count)
399 {
400     __asm__ ("cld\n\t"           // 清方向位。
401             "rep\n\t"           // 重复 ecx 指定的次数, 执行
402             "stosb"             // 将 al 中字符存入 es:[edi]中, 并且 edi++。
403             : "a" (c), "D" (s), "c" (count)
404             : "cx", "di");
405 return s;
406 }
407 #endif
408

```

11.12 termios.h 文件

11.12.1 功能描述

该文件含有终端 I/O 接口定义。包括 termios 数据结构和一些对通用终端接口设置的函数原型。这些函数用来读取或设置终端的属性、线路控制、读取或设置波特率以及读取或设置终端前端进程的组 id。虽然这是 linux 早期的头文件, 但已完全符合目前的 POSIX 标准, 并作了适当的扩展。

在该文件中定义的两个终端数据结构 termio 和 termios 是分别属于两类 UNIX 系列(或克隆), termio 是在 AT&T 系统 V 中定义的, 而 termios 是 POSIX 标准指定的。两个结构基本一样, 只是 termio 使用短整数类型定义模式标志集, 而 termios 使用长整数定义模式标志集。由于目前这两种结构都在使用, 因此为了兼容性, 大多数系统都同时支持它们。另外, 以前使用的是一类似的 sgtty 结构, 目前已基本不用。

11.12.2 代码注释

列表 linux/include/termios.h 文件

```

1 #ifndef TERMIOS\_H
2 #define TERMIOS\_H
3
4 #define TTY\_BUF\_SIZE 1024           // tty 中的缓冲区长度。
5
6 /* 0x54 is just a magic number to make these relatively unique ('T') */

```



```

/* 0x54 只是一个魔数，目的是为了使这些常数唯一('T') */
7 // tty 设备的 ioctl 调用命令集。ioctl 将命令编码在低位字中。
// 下面名称 TC[*] 的含义是 tty 控制命令。
// 取相应终端 termios 结构中的信息(参见 tcgetattr())。
8 #define TCGETS 0x5401
// 设置相应终端 termios 结构中的信息(参见 tcsetattr(), TCSANOW)。
9 #define TCSETS 0x5402
// 在设置终端 termios 的信息之前，需要先等待输出队列中所有数据处理完(耗尽)。对于修改参数
// 会影响输出的情况，就需要使用这种形式(参见 tcsetattr(), TCSADRAIN 选项)。
10 #define TCSETSW 0x5403
// 在设置 termios 的信息之前，需要先等待输出队列中所有数据处理完，并且刷新(清空)输入队列。
// 再设置(参见 tcsetattr(), TCSAFLUSH 选项)。
11 #define TCSETSF 0x5404
// 取相应终端 termio 结构中的信息(参见 tcgetattr())。
12 #define TCGETA 0x5405
// 设置相应终端 termio 结构中的信息(参见 tcsetattr(), TCSANOW 选项)。
13 #define TCSETA 0x5406
// 在设置终端 termio 的信息之前，需要先等待输出队列中所有数据处理完(耗尽)。对于修改参数
// 会影响输出的情况，就需要使用这种形式(参见 tcsetattr(), TCSADRAIN 选项)。
14 #define TCSETAW 0x5407
// 在设置 termio 的信息之前，需要先等待输出队列中所有数据处理完，并且刷新(清空)输入队列。
// 再设置(参见 tcsetattr(), TCSAFLUSH 选项)。
15 #define TCSETAF 0x5408
// 等待输出队列处理完毕(空)，如果参数值是 0，则发送一个 break(参见 tcsendbreak(), tcdrain())。
16 #define TCSBRK 0x5409
// 开始/停止控制。如果参数值是 0，则挂起输出；如果是 1，则重新开启挂起的输出；如果是 2，则挂
起
// 输入；如果是 3，则重新开启挂起的输入(参见 tcflow())。
17 #define TCXONC 0x540A
// 刷新已写输出但还没发送或已收但还没有读数据。如果参数是 0，则刷新(清空)输入队列；如果是 1，
// 则刷新输出队列；如果是 2，则刷新输入和输出队列(参见 tcflush())。
18 #define TCFLSH 0x540B
// 下面名称 TIOC[*] 的含义是 tty 输入输出控制命令。
// 设置终端串行线路专用模式。
19 #define TIOCEXCL 0x540C
// 复位终端串行线路专用模式。
20 #define TIOCNXCL 0x540D
// 设置 tty 为控制终端。(TIOCNOTTY - 禁止 tty 为控制终端)。
21 #define TIOCSCTTY 0x540E
// 读取指定终端设备进程的组 id(参见 tcgetpgrp())。
22 #define TIOCGPGRP 0x540F
// 设置指定终端设备进程的组 id(参见 tcsetpgrp())。
23 #define TIOCSPGRP 0x5410
// 返回输出队列中还未送出的字符数。
24 #define TIOCOUTQ 0x5411
// 模拟终端输入。该命令以一个指向字符的指针作为参数，并假装该字符是在终端上键入的。用户必须
// 在该控制终端上具有超级用户权限或具有读许可权限。
25 #define TIOCSTI 0x5412
// 读取终端设备窗口大小信息(参见 winsize 结构)。
26 #define TIOCGWINSZ 0x5413
// 设置终端设备窗口大小信息(参见 winsize 结构)。
27 #define TIOCSWINSZ 0x5414

```

```

// 返回 modem 状态控制引线的当前状态比特位标志集（参见下面 185-196 行）。
28 #define TIOCMGET 0x5415
// 设置单个 modem 状态控制引线的状态(true 或 false)(Individual control line Set)。
29 #define TIOCMBS 0x5416
// 复位单个 modem 状态控制引线的状态(Individual control line clear)。
30 #define TIOCMBS 0x5417
// 设置 modem 状态引线的状态。如果某一比特位置位，则 modem 对应的状态引线将置为有效。
31 #define TIOCMSET 0x5418
// 读取软件载波检测标志(1 - 开启；0 - 关闭)。
// 对于本地连接的终端或其它设备，软件载波标志是开启的，对于使用 modem 线路的终端或设备则
// 是关闭的。为了能使用这两个 ioctl 调用，tty 线路应该是以 O_NDELAY 方式打开的，这样 open()
// 就不会等待载波。
32 #define TIOCGSOFTCAR 0x5419
// 设置软件载波检测标志(1 - 开启；0 - 关闭)。
33 #define TIOCSSOFTCAR 0x541A
// 返回输入队列中还未取走字符的数目。
34 #define TIOCINQ 0x541B
35
// 窗口大小(Window size)属性结构。在窗口环境中可用于基于屏幕的应用程序。
// ioctl 中的 TIOCGWINSZ 和 TIOCSWINSZ 可用来读取或设置这些信息。
36 struct winsize {
37     unsigned short ws_row; // 窗口字符行数。
38     unsigned short ws_col; // 窗口字符列数。
39     unsigned short ws_xpixel; // 窗口宽度，像素值。
40     unsigned short ws_ypixel; // 窗口高度，像素值。
41 };
42
// AT&T 系统 V 的 termio 结构。
43 #define NCC 8 // termio 结构中控制字符数组的长度。
44 struct termio {
45     unsigned short c_iflag; /* input mode flags */ // 输入模式标志。
46     unsigned short c_oflag; /* output mode flags */ // 输出模式标志。
47     unsigned short c_cflag; /* control mode flags */ // 控制模式标志。
48     unsigned short c_lflag; /* local mode flags */ // 本地模式标志。
49     unsigned char c_line; /* line discipline */ // 线路规程（速率）。
50     unsigned char c_cc[NCC]; /* control characters */ // 控制字符数组。
51 };
52
// POSIX 的 termios 结构。
53 #define NCCS 17 // termios 结构中控制字符数组的长度。
54 struct termios {
55     unsigned long c_iflag; /* input mode flags */ // 输入模式标志。
56     unsigned long c_oflag; /* output mode flags */ // 输出模式标志。
57     unsigned long c_cflag; /* control mode flags */ // 控制模式标志。
58     unsigned long c_lflag; /* local mode flags */ // 本地模式标志。
59     unsigned char c_line; /* line discipline */ // 线路规程（速率）。
60     unsigned char c_cc[NCCS]; /* control characters */ // 控制字符数组。
61 };
62
63 /* c_cc characters */ /* c_cc 数组中的字符 */
// 以下是 c_cc 数组对应字符的索引值。
64 #define VINTR 0 // c_cc[VINTR] = INTR (^C), \003, 中断字符。
65 #define VQUIT 1 // c_cc[VQUIT] = QUIT (^), \034, 退出字符。

```



```

66 #define VERASE 2          // c_cc[VERASE] = ERASE (^H), \177, 擦出字符。
67 #define VKILL 3          // c_cc[VKILL] = KILL (^U), \025, 终止字符。
68 #define VEOF 4           // c_cc[VEOF] = EOF (^D), \004, 文件结束字符。
69 #define VTIME 5          // c_cc[VTIME] = TIME (\0), \0, 定时器值(参见后面说明)。
70 #define VMIN 6           // c_cc[VMIN] = MIN (\1), \1, 定时器值。
71 #define VSWTC 7          // c_cc[VSWTC] = SWTC (\0), \0, 交换字符。
72 #define VSTART 8         // c_cc[VSTART] = START (^Q), \021, 开始字符。
73 #define VSTOP 9          // c_cc[VSTOP] = STOP (^S), \023, 停止字符。
74 #define VSUSP 10         // c_cc[VSUSP] = SUSP (^Z), \032, 挂起字符。
75 #define VEOL 11          // c_cc[VEOL] = EOL (\0), \0, 行结束字符。
76 #define VREPRINT 12      // c_cc[VREPRINT] = REPRINT (^R), \022, 重显示字符。
77 #define VDISCARD 13      // c_cc[VDISCARD] = DISCARD (^O), \017, 丢弃字符。
78 #define VWERASE 14       // c_cc[VWERASE] = WERASE (^W), \027, 单词擦除字符。
79 #define VLNEXT 15        // c_cc[VLNEXT] = LNEXT (^V), \026, 下一行字符。
80 #define VEOL2 16         // c_cc[VEOL2] = EOL2 (\0), \0, 行结束 2。
81
82 /* c_iflag bits */      /* c_iflag 比特位 */
// termios 结构输入模式字段 c_iflag 各种标志的符号常数。
83 #define IGNBRK 0000001   // 输入时忽略 BREAK 条件。
84 #define BRKINT 0000002   // 在 BREAK 时产生 SIGINT 信号。
85 #define IGNPAR 0000004   // 忽略奇偶校验出错的字符。
86 #define PARMRK 0000010   // 标记奇偶校验错。
87 #define INPCK 0000020    // 允许输入奇偶校验。
88 #define ISTRIP 0000040    // 屏蔽字符第 8 位。
89 #define INLCR 0000100    // 输入时将换行符 NL 映射成回车符 CR。
90 #define IGNCR 0000200    // 忽略回车符 CR。
91 #define ICRNL 0000400    // 在输入时将回车符 CR 映射成换行符 NL。
92 #define IUCLC 0001000    // 在输入时将大写字母转换成小写字母。
93 #define IXON 0002000     // 允许开始/停止 (XON/XOFF) 输出控制。
94 #define IXANY 0004000    // 允许任何字符重启输出。
95 #define IXOFF 0010000    // 允许开始/停止 (XON/XOFF) 输入控制。
96 #define IMAXBEL 0020000  // 输入队列满时响铃。
97
98 /* c_oflag bits */      /* c_oflag 比特位 */
// termios 结构中输出模式字段 c_oflag 各种标志的符号常数。
99 #define OPOST 0000001    // 执行输出处理。
100 #define OLCUC 0000002    // 在输出时将小写字母转换成大写字母。
101 #define ONLCR 0000004    // 在输出时将换行符 NL 映射成回车-换行符 CR-NL。
102 #define OCRNL 0000010    // 在输出时将回车符 CR 映射成换行符 NL。
103 #define ONOCR 0000020    // 在 0 列不输出回车符 CR。
104 #define ONLRET 0000040   // 换行符 NL 执行回车符的功能。
105 #define OFILL 0000100    // 延迟时使用填充字符而不使用时间延迟。
106 #define OFDEL 0000200    // 填充字符是 ASCII 码 DEL。如果未设置, 则使用 ASCII NULL。
107 #define NLDLY 0000400    // 选择换行延迟。
108 #define NL0 0000000      // 换行延迟类型 0。
109 #define NL1 0000400      // 换行延迟类型 1。
110 #define CRDLY 0003000    // 选择回车延迟。
111 #define CR0 0000000      // 回车延迟类型 0。
112 #define CR1 0001000      // 回车延迟类型 1。
113 #define CR2 0002000      // 回车延迟类型 2。
114 #define CR3 0003000      // 回车延迟类型 3。
115 #define TABDLY 0014000   // 选择水平制表延迟。
116 #define TAB0 0000000     // 水平制表延迟类型 0。

```

```

117 #define TAB1 0004000 // 水平制表延迟类型 1。
118 #define TAB2 0010000 // 水平制表延迟类型 2。
119 #define TAB3 0014000 // 水平制表延迟类型 3。
120 #define XTABS 0014000 // 将制表符 TAB 换成空格，该值表示空格数。
121 #define BSDLY 0020000 // 选择退格延迟。
122 #define BS0 0000000 // 退格延迟类型 0。
123 #define BS1 0020000 // 退格延迟类型 1。
124 #define VTDLY 0040000 // 纵向制表延迟。
125 #define VT0 0000000 // 纵向制表延迟类型 0。
126 #define VT1 0040000 // 纵向制表延迟类型 1。
127 #define FFDLY 0040000 // 选择换页延迟。
128 #define FF0 0000000 // 换页延迟类型 0。
129 #define FF1 0040000 // 换页延迟类型 1。
130
131 /* c_cflag bit meaning */ /* c_cflag 比特位的含义 */
// termios 结构中控制模式标志字段 c_cflag 标志的符号常数 (8 进制数)。
132 #define CBAUD 0000017 // 传输速率位屏蔽码。
133 #define B0 0000000 /* hang up */ /* 挂断线路 */
134 #define B50 0000001 // 波特率 50。
135 #define B75 0000002 // 波特率 75。
136 #define B110 0000003 // 波特率 110。
137 #define B134 0000004 // 波特率 134。
138 #define B150 0000005 // 波特率 150。
139 #define B200 0000006 // 波特率 200。
140 #define B300 0000007 // 波特率 300。
141 #define B600 0000010 // 波特率 600。
142 #define B1200 0000011 // 波特率 1200。
143 #define B1800 0000012 // 波特率 1800。
144 #define B2400 0000013 // 波特率 2400。
145 #define B4800 0000014 // 波特率 4800。
146 #define B9600 0000015 // 波特率 9600。
147 #define B19200 0000016 // 波特率 19200。
148 #define B38400 0000017 // 波特率 38400。
149 #define EXTA B19200 // 扩展波特率 A。
150 #define EXTB B38400 // 扩展波特率 B。

151 #define CSIZE 0000060 // 字符位宽度屏蔽码。
152 #define CS5 0000000 // 每字符 5 比特位。
153 #define CS6 0000020 // 每字符 6 比特位。
154 #define CS7 0000040 // 每字符 7 比特位。
155 #define CS8 0000060 // 每字符 8 比特位。
156 #define CSTOPB 0000100 // 设置两个停止位，而不是 1 个。
157 #define CREAD 0000200 // 允许接收。
158 #define CPARENB 0000400 // 开启输出时产生奇偶位、输入时进行奇偶校验。
159 #define CPARODD 0001000 // 输入/输入校验是奇校验。
160 #define HUPCL 0002000 // 最后进程关闭后挂断。
161 #define CLOCAL 0004000 // 忽略调制解调器(modem)控制线路。
162 #define CIBAUD 03600000 /* input baud rate (not used) */ /* 输入波特率(未使用) */
163 #define CRTSCTS 020000000000 /* flow control */ /* 流控制 */
164
165 #define PARENB CPARENB // 开启输出时产生奇偶位、输入时进行奇偶校验。
166 #define PARODD CPARODD // 输入/输入校验是奇校验。
167

```

```

168 /* c_lflag bits */          /* c_lflag 比特位 */
    // termios 结构中本地模式标志字段 c_lflag 的符号常数。
169 #define ISIG      0000001    // 当收到字符 INTR、QUIT、SUSP 或 DSUSP，产生相应的信号。
170 #define ICANON    0000002    // 开启规范模式（熟模式）。
171 #define XCASE     0000004    // 若设置了 ICANON，则终端是大写字母的。
172 #define ECHO      0000010    // 回显输入字符。
173 #define ECHOE     0000020    // 若设置了 ICANON，则 ERASE/WERASE 将擦除前一字符/单词。
174 #define ECHOK     0000040    // 若设置了 ICANON，则 KILL 字符将擦除当前行。
175 #define ECHONL    0000100    // 如设置了 ICANON，则即使 ECHO 没有开启也回显 NL 字符。
176 #define NOFLSH    0000200    // 当生成 SIGINT 和 SIGQUIT 信号时不刷新输入输出队列，当
    // 生成 SIGSUSP 信号时，刷新输入队列。
177 #define TOSTOP    0000400    // 发送 SIGTTOU 信号到后台进程的进程组，该后台进程试图写
    // 自己的控制终端。
178 #define ECHOCTL   0001000    // 若设置了 ECHO，则除 TAB、NL、START 和 STOP 以外的 ASCII
    // 控制信号将被回显成象^X 式样，X 值是控制符+0x40。
179 #define ECHOPRT   0002000    // 若设置了 ICANON 和 IECHO，则字符在擦除时将显示。
180 #define ECHOKE    0004000    // 若设置了 ICANON，则 KILL 通过擦除行上的所有字符被回显。
181 #define FLUSHO    0010000    // 输出被刷新。通过键入 DISCARD 字符，该标志被翻转。
182 #define PENDIN    0040000    // 当下一个字符是读时，输入队列中的所有字符将被重显。
183 #define IEXTEN    0100000    // 开启实现时定义的输入处理。
184
185 /* modem lines */          /* modem 线路信号符号常数 */
186 #define TIOCM_LE   0x001    // 线路允许(Line Enable)。
187 #define TIOCM_DTR  0x002    // 数据终端就绪(Data Terminal Ready)。
188 #define TIOCM_RTS  0x004    // 请求发送(Request to Send)。
189 #define TIOCM_ST    0x008    // 串行数据发送(Serial Transfer)。[??]
190 #define TIOCM_SR    0x010    // 串行数据接收(Serial Receive)。[??]
191 #define TIOCM_CTS   0x020    // 清除发送(Clear To Send)。
192 #define TIOCM_CAR   0x040    // 载波监测(Carrier Detect)。
193 #define TIOCM_RNG   0x080    // 响铃指示(Ring indicate)。
194 #define TIOCM_DSR   0x100    // 数据设备就绪(Data Set Ready)。
195 #define TIOCM_CD    TIOCM_CAR
196 #define TIOCM_RI    TIOCM_RNG
197
198 /* tcflow() and TCXONC use these */ /* tcflow() 和 TCXONC 使用这些符号常数 */
199 #define TCOOFF      0        // 挂起输出。
200 #define TCOON       1        // 重启被挂起的输出。
201 #define TCIOFF      2        // 系统传输一个 STOP 字符，使设备停止向系统传输数据。
202 #define TCION       3        // 系统传输一个 START 字符，使设备开始向系统传输数据。
203
204 /* tcflush() and TCFLSH use these */ /* tcflush() 和 TCFLSH 使用这些符号常数 */
205 #define TCIFLUSH    0        // 清接收到的数据但不读。
206 #define TCOFLUSH    1        // 清已写的数据但不传送。
207 #define TCIOFLUSH   2        // 清接收到的数据但不读。清已写的数据但不传送。
208
209 /* tcsetattr uses these */          /* tcsetattr() 使用这些符号常数 */
210 #define TCSANOW     0        // 改变立即发生。
211 #define TCSADRAIN   1        // 改变在所有已写的输出被传输之后发生。
212 #define TCSAFLUSH   2        // 改变在所有已写的输出被传输之后并且在所有接收到但
    // 还没有读取的数据被丢弃之后发生。
213
214 typedef int speed_t;          // 波特率数值类型。
215

```

```

// 返回 termios_p 所指 termios 结构中的接收波特率。
216 extern speed_t cfgetispeed(struct termios *termios_p);
// 返回 termios_p 所指 termios 结构中的发送波特率。
217 extern speed_t cfgetospeed(struct termios *termios_p);
// 将 termios_p 所指 termios 结构中的接收波特率设置为 speed。
218 extern int cfsetispeed(struct termios *termios_p, speed_t speed);
// 将 termios_p 所指 termios 结构中的发送波特率设置为 speed。
219 extern int cfsetospeed(struct termios *termios_p, speed_t speed);
// 等待 fildes 所指对象已写输出数据被传送出去。
220 extern int tcdrain(int fildes);
// 挂起/重启 fildes 所指对象数据的接收和发送。
221 extern int tcflow(int fildes, int action);
// 丢弃 fildes 指定对象所有已写但还没传送以及所有已收到但还没有读取的数据。
222 extern int tcflush(int fildes, int queue_selector);
// 获取与句柄 fildes 对应对象的参数，并将其保存在 termios_p 所指的地方。
223 extern int tcgetattr(int fildes, struct termios *termios_p);
// 如果终端使用异步串行数据传输，则在一定时间内连续传输一系列 0 值比特位。
224 extern int tcsendbreak(int fildes, int duration);
// 使用 termios 结构指针 termios_p 所指的数据，设置与终端相关的参数。
225 extern int tcsetattr(int fildes, int optional_actions,
226                     struct termios *termios_p);
227
228 #endif
229

```

11.12.3 其它信息

控制字符 TIME、MIN

在非规范模式输入处理中，输入字符没有被处理成行，因此擦除和终止处理也就不会发生。MIN 和 TIMEDE 的值即用于确定如何处理接收到的字符。

MIN 表示当满足读操作时（也即，当字符返给用户时）需要读取的最少字符数。TIME 是以 1/10 秒计数的定时值，用于超时定时和短期数据传输。这两个字符的四种组合情况及其相互作用描述如下：

MIN > 0, TIME > 0 的情况：

在这种情况下，TIME 起字符与字符间的定时器作用，并在接收到第 1 个字符后开始起作用。由于它是字符与字符间的定时器，所以在每收到一个字符就会被复位重启。MIN 与 TIME 之间的相互作用如下：一旦收到一个字符，字符间定时器就开始工作。如果在定时器超时（注意定时器每收到一个字符就会重新开始计时）之前收到了 MIN 个字符，则读操作即被满足。如果在 MIN 个字符被收到之前定时器超时了，就将到此时已收到的字符返回给用户。注意，如果 TIME 超时，则起码有一个接收到的字符将被返回，因为定时器只有在接收到了一个字符之后才开始起作用（计时）。在这种情况下（MIN > 0, TIME > 0），读操作将会睡眠，直到接收到第 1 个字符激活 MIN 与 TIME 机制。如果读到字符数少于已有的字符数，那么定时器将不会被重新激活，因而随后的读操作将被立刻满足。

MIN > 0, TIME = 0 的情况：

在这种情况下，由于 TIME 的值是 0，因此定时器不起作用，只有 MIN 是有意义的。等待的读操作只有当接收到 MIN 个字符时才会被满足（等待着的操作将睡眠直到收到 MIN 个字符）。使用这种情况去读基于记录的终端 I/O 的程序将会在读操作中被不确定地（随意地）阻塞。

MIN = 0, TIME > 0 的情况:

在这种情况下, 由于 MIN=0, 则 TIME 不再起字符间的定时器作用, 而是一个读操作定时器, 并在读操作一开始就起作用。只要接收到一个字符或者定时器超时就已满足读操作。注意, 在这种情况下, 如果定时器超时了, 将读不到一个字符。如果定时器没有超时, 那么只有在读到一个字符之后读操作才会满足。因此在这种情况下, 读操作不会无限制地(不确定地)被阻塞, 以等待字符。在读操作开始后, 如果在 TIME*0.10 秒的时间内没有收到字符, 读操作将以收到 0 个字符而返回。

MIN = 0, TIME = 0 的情况:

在这种情况下, 读操作会立刻返回。所请求读的字符数或缓冲队列中现有字符数中的最小值将被返回, 而不会等待更多的字符被输入缓冲中。

总的来说, 在非规范模式下, 这两个值是超时定时值和字符计数值。MIN 表示为了满足读操作, 需要读取的最少字符数。TIME 是一个十分之一秒计数的计时值。当这两个都设置的话, 读操作将等待, 直到至少读到一个字符, 然后在以读取 MIN 个字符或者时间 TIME 在读取最后一个字符后超时。如果仅设置了 MIN, 那么在读取 MIN 个字符之前读操作将不返回。如果仅设置了 TIME, 那么在读到至少一个字符或者定时超时后读操作将立刻返回。如果两个都没有设置, 则读操作将立刻返回, 仅给出目前已读的字节数。

11.13 time.h 文件

11.13.1 功能描述

该头文件用于涉及处理时间的函数。在 MINIX 中有一段对时间的描述很有趣: 时间的处理较为复杂, 比如什么是 GMT (格林威治标准时间)、本地时间或其它时间等。尽管主教 Ussher (1581-1656 年) 曾经计算过, 根据圣经, 世界开始之日是公元前 4004 年 10 月 12 日上午 9 点, 但在 UNIX 世界里, 时间是从 GMT 1970 年 1 月 1 日午夜开始的, 在这之前, 所有均是空无的和(无效的)。

11.13.2 代码注释

列表 linux/include/time.h 文件

```

1 #ifndef \_TIME\_H
2 #define \_TIME\_H
3
4 #ifndef \_TIME\_T
5 #define \_TIME\_T
6 typedef long time\_t;           // 从 GMT 1970 年 1 月 1 日开始的以秒计数的时间(日历时间)。
7 #endif
8
9 #ifndef \_SIZE\_T
10 #define \_SIZE\_T
11 typedef unsigned int size\_t;
12 #endif
13
14 #define CLOCKS\_PER\_SEC 100      // 系统时钟滴答频率, 100HZ。

```



```

15
16 typedef long clock\_t; // 从进程开始系统经过的时钟滴答数。
17
18 struct tm {
19     int tm_sec; // 秒数 [0, 59]。
20     int tm_min; // 分钟数 [0, 59]。
21     int tm_hour; // 小时数 [0, 59]。
22     int tm_mday; // 1 个月的天数 [0, 31]。
23     int tm_mon; // 1 年中月份 [0, 11]。
24     int tm_year; // 从 1900 年开始的年数。
25     int tm_wday; // 1 星期中的某天 [0, 6] (星期天 =0)。
26     int tm_yday; // 1 年中的某天 [0, 365]。
27     int tm_isdst; // 夏令时标志。
28 };
29
// 以下是有关时间操作的函数原型。
// 确定处理器使用时间。返回程序所用处理器时间（滴答数）的近似值。
30 clock\_t clock(void);
// 取时间（秒数）。返回从 1970.1.1:0:0 开始的秒数（称为日历时间）。
31 time\_t time(time\_t * tp);
// 计算时间差。返回时间 time2 与 time1 之间经过的秒数。
32 double difftime(time\_t time2, time\_t time1);
// 将 tm 结构表示的时间转换成日历时间。
33 time\_t mktime(struct tm * tp);
34
// 将 tm 结构表示的时间转换成一个字符串。返回指向该串的指针。
35 char * asctime(const struct tm * tp);
// 将日历时间转换成一个字符串形式，如 “Wed Jun 30 21:49:08:1993\n”。
36 char * ctime(const time\_t * tp);
// 将日历时间转换成 tm 结构表示的 UTC 时间（UTC - 世界时间代码 Universal Time Code）。
37 struct tm * gmtime(const time\_t * tp);
// 将日历时间转换成 tm 结构表示的指定时间区 (timezone) 的时间。
38 struct tm * localtime(const time\_t * tp);
// 将 tm 结构表示的时间利用格式字符串 fmt 转换成最大长度为 smax 的字符串并将结果存储在 s 中。
39 size\_t strftime(char * s, size\_t smax, const char * fmt, const struct tm * tp);
// 初始化时间转换信息，使用环境变量 TZ，对 zname 变量进行初始化。
// 在与时间区相关的时间转换函数中将自动调用该函数。
40 void tzset(void);
41
42 #endif
43

```

11.14 unistd.h 文件

11.14.1 功能描述

11.14.2 代码注释

列表 linux/include/unistd.h 文件

```

1 #ifndef UNISTD\_H
2 #define UNISTD\_H
3
4 /* ok, this may be a joke, but I'm working on it */
   /* ok, 这也许是个玩笑, 但我正在着手处理 */
   // 下面符号常数指出符合 IEEE 标准 1003.1 实现的版本号, 是一个整数值。
5 #define POSIX\_VERSION 198808L
6
   // chown() 和 fchown() 的使用受限于进程的权限。/* 只有超级用户可以执行 chown (我想..) */
7 #define POSIX\_CHOWN\_RESTRICTED /* only root can do a chown (I think..) */
   // 长于 (NAME_MAX) 的路径名将产生错误, 而不会自动截断。/* 路径名不截断 (但是请看内核代码) */
8 #define POSIX\_NO\_TRUNC /* no pathname truncation (but see in kernel) */
   // 下面这个符号将定义成字符值, 该值将禁止终端对其的处理。/* 禁止象 ^C 这样的字符 */
9 #define POSIX\_VDISABLE '\0' /* character to disable things like ^C */
   // 每个进程都有一保存的 set-user-ID 和一保存的 set-group-ID。/* 我们将着手对此进行处理 */
10 ##define _POSIX_SAVED_IDS */ /* we'll get to this yet */
   // 系统实现支持作业控制。/* 我们还没有支持这项标准, 希望很快就行 */
11 ##define _POSIX_JOB_CONTROL */ /* we aren't there quite yet. Soon hopefully */
12
13 #define STDIN\_FILENO 0 // 标准输入文件句柄 (描述符) 号。
14 #define STDOUT\_FILENO 1 // 标准输出文件句柄号。
15 #define STDERR\_FILENO 2 // 标准出错文件句柄号。
16
17 #ifndef NULL
18 #define NULL ((void *)0) // 定义空指针。
19 #endif
20
21 /* access */ /* 文件访问 */
   // 以下定义的符号常数用于 access() 函数。
22 #define F\_OK 0 // 检测文件是否存在。
23 #define X\_OK 1 // 检测是否可执行 (搜索)。
24 #define W\_OK 2 // 检测是否可写。
25 #define R\_OK 4 // 检测是否可读。
26
27 /* lseek */ /* 文件指针重定位 */
   // 以下符号常数用于 lseek() 和 fcntl() 函数。
28 #define SEEK\_SET 0 // 将文件读写指针设置为偏移值。
29 #define SEEK\_CUR 1 // 将文件读写指针设置为当前值加上偏移值。
30 #define SEEK\_END 2 // 将文件读写指针设置为文件长度加上偏移值。
31
32 /* _SC stands for System Configuration. We don't use them much */
   /* _SC 表示系统配置。我们很少使用 */
   // 下面的符号常数用于 sysconf() 函数。
33 #define SC\_ARG\_MAX 1 // 最大变量数。
34 #define SC\_CHILD\_MAX 2 // 子进程最大数。
35 #define SC\_CLOCKS\_PER\_SEC 3 // 每秒滴答数。
36 #define SC\_NGROUPS\_MAX 4 // 最大组数。
37 #define SC\_OPEN\_MAX 5 // 最大打开文件数。
38 #define SC\_JOB\_CONTROL 6 // 作业控制。
39 #define SC\_SAVED\_IDS 7 // 保存的标识符。

```

```

40 #define SC_VERSION          8    // 版本。
41
42 /* more (possibly) configurable things - now pathnames */
43 /* 更多的（可能的）可配置参数 - 现在用于路径名 */
44 // 下面的符号常数用于 pathconf() 函数。
45 #define PC_LINK_MAX        1    // 连接最大数。
46 #define PC_MAX_CANON      2    // 最大常规文件数。
47 #define PC_MAX_INPUT      3    // 最大输入长度。
48 #define PC_NAME_MAX       4    // 名称最大长度。
49 #define PC_PATH_MAX       5    // 路径最大长度。
50 #define PC_PIPE_BUF       6    // 管道缓冲大小。
51 #define PC_NO_TRUNC       7    // 文件名不截断。
52 #define PC_VDISABLE      8    //
53 #define PC_CHOWN_RESTRICTED 9    // 改变宿主受限。
54
55 #include <sys/stat.h>    // 文件状态头文件。含有文件或文件系统状态结构 stat{} 和常量。
56 #include <sys/times.h>  // 定义了进程中运行时间结构 tms 以及 times() 函数原型。
57 #include <sys/utsname.h> // 系统名称结构头文件。
58 #include <utime.h>      // 用户时间头文件。定义了访问和修改时间结构以及 utime() 原型。
59
60 #ifdef LIBRARY
61 // 以下是内核实现的系统调用符号常数，用于作为系统调用函数表中的索引值。
62 ( include/linux/sys.h )
63 #define NR_setup      0    /* used only by init, to get system going */
64                               /* __NR_setup 仅用于初始化，以启动系统 */
65 #define NR_exit      1
66 #define NR_fork      2
67 #define NR_read      3
68 #define NR_write     4
69 #define NR_open      5
70 #define NR_close     6
71 #define NR_waitpid   7
72 #define NR_creat     8
73 #define NR_link      9
74 #define NR_unlink   10
75 #define NR_execve   11
76 #define NR_chdir    12
77 #define NR_time     13
78 #define NR_mknod    14
79 #define NR_chmod    15
80 #define NR_chown    16
81 #define NR_break    17
82 #define NR_stat     18
83 #define NR_lseek    19
84 #define NR_getpid   20
85 #define NR_mount    21
86 #define NR_umount   22
87 #define NR_setuid   23
88 #define NR_getuid   24
89 #define NR_stime    25
90 #define NR_ptrace   26
91 #define NR_alarm    27

```

88	<code>#define</code>	NR_fstat	28
89	<code>#define</code>	NR_pause	29
90	<code>#define</code>	NR_utime	30
91	<code>#define</code>	NR_stty	31
92	<code>#define</code>	NR_gtty	32
93	<code>#define</code>	NR_access	33
94	<code>#define</code>	NR_nice	34
95	<code>#define</code>	NR_ftime	35
96	<code>#define</code>	NR_sync	36
97	<code>#define</code>	NR_kill	37
98	<code>#define</code>	NR_rename	38
99	<code>#define</code>	NR_mkdir	39
100	<code>#define</code>	NR_rmdir	40
101	<code>#define</code>	NR_dup	41
102	<code>#define</code>	NR_pipe	42
103	<code>#define</code>	NR_times	43
104	<code>#define</code>	NR_prof	44
105	<code>#define</code>	NR_brk	45
106	<code>#define</code>	NR_setgid	46
107	<code>#define</code>	NR_getgid	47
108	<code>#define</code>	NR_signal	48
109	<code>#define</code>	NR_geteuid	49
110	<code>#define</code>	NR_getegid	50
111	<code>#define</code>	NR_acct	51
112	<code>#define</code>	NR_phys	52
113	<code>#define</code>	NR_lock	53
114	<code>#define</code>	NR_ioctl	54
115	<code>#define</code>	NR_fcntl	55
116	<code>#define</code>	NR_mpx	56
117	<code>#define</code>	NR_setpgid	57
118	<code>#define</code>	NR_ulimit	58
119	<code>#define</code>	NR_uname	59
120	<code>#define</code>	NR_umask	60
121	<code>#define</code>	NR_chroot	61
122	<code>#define</code>	NR_ustat	62
123	<code>#define</code>	NR_dup2	63
124	<code>#define</code>	NR_getppid	64
125	<code>#define</code>	NR_getpgrp	65
126	<code>#define</code>	NR_setsid	66
127	<code>#define</code>	NR_sigaction	67
128	<code>#define</code>	NR_sgetmask	68
129	<code>#define</code>	NR_ssetmask	69
130	<code>#define</code>	NR_setreuid	70
131	<code>#define</code>	NR_setregid	71
132			

// 以下定义系统调用嵌入式汇编宏函数。

// 不带参数的系统调用宏函数。type name(void)。

// %0 - eax(__res), %1 - eax(__NR_##name)。其中 name 是系统调用的名称，与 __NR_ 组合形成上面的系统调用符号常数，从而用来对系统调用表中函数指针寻址。

// 返回：如果返回值大于等于 0，则返回该值，否则置出错号 errno，并返回-1。

```

133 #define syscall0(type,name) \
134 type name(void) \
135 { \

```

```

136 long __res; \
137 __asm__ volatile ("int $0x80" \           // 调用系统中断 0x80。
138                  : "=a" (__res) \         // 返回值→eax(__res)。
139                  : "" (__NR_##name)); \    // 输入为系统中断调用号 __NR_name。
140 if (__res >= 0) \                          // 如果返回值>=0, 则直接返回该值。
141     return (type) __res; \
142 errno = -__res; \                         // 否则置出错号, 并返回-1。
143 return -1; \
144 }
145
// 有 1 个参数的系统调用宏函数。type name(atype a)
// %0 - eax(__res), %1 - eax(__NR_name), %2 - ebx(a)。
146 #define __syscall1(type, name, atype, a) \
147 type name(atype a) \
148 { \
149     long __res; \
150     __asm__ volatile ("int $0x80" \
151                      : "=a" (__res) \
152                      : "" (__NR_##name), "b" ((long) (a))); \
153     if (__res >= 0) \
154         return (type) __res; \
155     errno = -__res; \
156     return -1; \
157 }
158
// 有 2 个参数的系统调用宏函数。type name(atype a, btype b)
// %0 - eax(__res), %1 - eax(__NR_name), %2 - ebx(a), %3 - ecx(b)。
159 #define __syscall2(type, name, atype, a, btype, b) \
160 type name(atype a, btype b) \
161 { \
162     long __res; \
163     __asm__ volatile ("int $0x80" \
164                      : "=a" (__res) \
165                      : "" (__NR_##name), "b" ((long) (a)), "c" ((long) (b))); \
166     if (__res >= 0) \
167         return (type) __res; \
168     errno = -__res; \
169     return -1; \
170 }
171
// 有 3 个参数的系统调用宏函数。type name(atype a, btype b, ctype c)
// %0 - eax(__res), %1 - eax(__NR_name), %2 - ebx(a), %3 - ecx(b), %4 - edx(c)。
172 #define __syscall3(type, name, atype, a, btype, b, ctype, c) \
173 type name(atype a, btype b, ctype c) \
174 { \
175     long __res; \
176     __asm__ volatile ("int $0x80" \
177                      : "=a" (__res) \
178                      : "" (__NR_##name), "b" ((long) (a)), "c" ((long) (b)), "d" ((long) (c))); \
179     if (__res >= 0) \
180         return (type) __res; \
181     errno = -__res; \
182     return -1; \

```

```

183 }
184
185 #endif /* __LIBRARY__ */
186
187 extern int errno; // 出错号，全局变量。
188
189 // 对应各系统调用的函数原型定义。
189 int access(const char * filename, mode\_t mode);
190 int acct(const char * filename);
191 int alarm(int sec);
192 int brk(void * end_data_segment);
193 void * sbrk(ptrdiff\_t increment);
194 int chdir(const char * filename);
195 int chmod(const char * filename, mode\_t mode);
196 int chown(const char * filename, uid\_t owner, gid\_t group);
197 int chroot(const char * filename);
198 int close(int fildes);
199 int creat(const char * filename, mode\_t mode);
200 int dup(int fildes);
201 int execve(const char * filename, char ** argv, char ** envp);
202 int execv(const char * pathname, char ** argv);
203 int execvp(const char * file, char ** argv);
204 int execl(const char * pathname, char * arg0, ...);
205 int execlp(const char * file, char * arg0, ...);
206 int execle(const char * pathname, char * arg0, ...);
207 volatile void exit(int status);
208 volatile void \_exit(int status);
209 int fcntl(int fildes, int cmd, ...);
210 int fork(void);
211 int getpid(void);
212 int getuid(void);
213 int geteuid(void);
214 int getgid(void);
215 int getegid(void);
216 int ioctl(int fildes, int cmd, ...);
217 int kill(pid\_t pid, int signal);
218 int link(const char * filename1, const char * filename2);
219 int lseek(int fildes, off\_t offset, int origin);
220 int mknod(const char * filename, mode\_t mode, dev\_t dev);
221 int mount(const char * specialfile, const char * dir, int rwflag);
222 int nice(int val);
223 int open(const char * filename, int flag, ...);
224 int pause(void);
225 int pipe(int * fildes);
226 int read(int fildes, char * buf, off\_t count);
227 int setpgrp(void);
228 int setpgid(pid\_t pid, pid\_t pgid);
229 int setuid(uid\_t uid);
230 int setgid(gid\_t gid);
231 void (*signal(int sig, void (*fn)(int)))(int);
232 int stat(const char * filename, struct stat * stat_buf);
233 int fstat(int fildes, struct stat * stat_buf);
234 int stime(time\_t * tptr);

```

```

235 int sync(void);
236 time_t time(time_t * tloc);
237 time_t times(struct tms * tbuf);
238 int ulimit(int cmd, long limit);
239 mode_t umask(mode_t mask);
240 int umount(const char * specialfile);
241 int uname(struct utsname * name);
242 int unlink(const char * filename);
243 int ustat(dev_t dev, struct ustat * ubuf);
244 int utime(const char * filename, struct utimbuf * times);
245 pid_t waitpid(pid_t pid, int * wait_stat, int options);
246 pid_t wait(int * wait_stat);
247 int write(int fildes, const char * buf, off_t count);
248 int dup2(int oldfd, int newfd);
249 int getppid(void);
250 pid_t getpgrp(void);
251 pid_t setsid(void);
252
253 #endif
254

```

11.15 utime.h 文件

11.15.1 功能描述

该文件定义了文件访问和修改时间结构 `utimbuf` 以及 `utime()` 函数原型。

11.15.2 代码注释

列表 linux/include/utime.h 文件





```

1 #ifndef _UTIME_H
2 #define _UTIME_H
3
4 #include <sys/types.h> /* I know - shouldn't do this, but .. */
5 /* 我知道 - 不应该这样做, 但是.. */
6 struct utimbuf {
7     time_t actime;      // 文件访问时间。从 1970.1.1:0:0:0 开始的秒数。
8     time_t modtime;     // 文件修改时间。从 1970.1.1:0:0:0 开始的秒数。
9 };
10 // 设置文件访问和修改时间函数。
11 extern int utime(const char *filename, struct utimbuf *times);
12
13 #endif
14

```

11.16 include/asm/目录下的文件

列表 linux/include/asm/目录下的文件

Name	Size	Last modified (GMT)	Description
 io.h	477 bytes	1991-08-07 10:17:51	m
 memory.h	507 bytes	1991-06-15 20:54:44	m
 segment.h	1366 bytes	1991-11-25 18:48:24	m
 system.h	1711 bytes	1991-09-17 13:08:31	m

11.17 io.h 文件

11.17.1 功能描述

该文件中定义了对硬件 I/O 端口访问的嵌入式汇编宏函数：outb()、inb() 以及 outb_p() 和 inb_p()。前面两个函数与后面两个的主要区别在于后者代码中使用了 jmp 指令进行了时间延迟。

11.17.2 代码注释

列表 linux/include/asm/io.h 文件

```

1  // 硬件端口字节输出函数。
2  // 参数：value - 欲输出字节；port - 端口。
3  1 #define outb(value,port) \
4  2 __asm__ ("outb %%al,%%dx"::"a" (value),"d" (port))
5
6  // 硬件端口字节输入函数。
7  // 参数：port - 端口。返回读取的字节。
8  5 #define inb(port) ({ \
9  6 unsigned char _v; \
10 7 __asm__ volatile ("inb %%dx,%%al"::"a" (_v):"d" (port)); \
11 8 _v; \
12 9 })
13
14 // 带延迟的硬件端口字节输出函数。
15 // 参数：value - 欲输出字节；port - 端口。
16 11 #define outb_p(value,port) \
17 12 __asm__ ("outb %%al,%%dx\n" \

```

```

13         "\tjmp 1f\n" \
14         "1:\tjmp 1f\n" \
15         "1:": "a" (value), "d" (port))
16
17     /// 带延迟的硬件端口字节输入函数。
18     // 参数: port - 端口。返回读取的字节。
19 #define inb_p(port) ({ \
20     unsigned char _v; \
21     __asm__ volatile ("inb %%dx, %%al\n" \
22         "\tjmp 1f\n" \
23         "1:\tjmp 1f\n" \
24         "1:": "=a" (_v): "d" (port)); \
25     _v; \
26 })

```

11.18 memory.h 文件

11.18.1 功能描述

该文件含有一个内存复制嵌入式汇编宏 `memcpy()`。与 `string.h` 中定义的 `memcpy()` 相同，只是后者采用的是嵌入式汇编 C 函数形式定义的。

11.18.2 代码注释

列表 linux/include/asm/memory.h 文件

```

1  /*
2  * NOTE!!! memcpy(dest, src, n) assumes ds=es=normal data segment. This
3  * goes for all kernel functions (ds=es=kernel space, fs=local data,
4  * gs=null), as well as for all well-behaving user programs (ds=es=
5  * user data space). This is NOT a bug, as any user program that changes
6  * es deserves to die if it isn't careful.
7  */
8  /*
9  * 注意!!!memcpy(dest, src, n)假设段寄存器 ds=es=通常数据段。在内核中使用的
10 * 所有函数都基于该假设 (ds=es=内核空间, fs=局部数据空间, gs=null), 具有良好
11 * 行为的应用程序也是这样 (ds=es=用户数据空间)。如果任何用户程序随意改动了
12 * es 寄存器而出错, 则并不是由于系统程序错误造成的。
13 */
14 /// 内存块复制。从源地址 src 处开始复制 n 个字节到目的地址 dest 处。
15 // 参数: dest - 复制的目的地址, src - 复制的源地址, n - 复制字节数。
16 // %0 - edi(目的地址 dest), %1 - esi(源地址 src), %2 - ecx(字节数 n),
17 #define memcpy(dest, src, n) ({ \
18     void * _res = dest; \
19     __asm__ ("cld;rep;movsb" \
20         // 从 ds:[esi]复制到 es:[edi], 并且 esi++, edi++.
21         // 共复制 ecx(n) 字节。
22         :: "D" ((long) (_res)), "S" ((long) (src)), "c" ((long) (n)) \
23         : "di", "si", "cx"); \
24     _res; \
25 })

```

14 })
 15

11.19 segment.h 文件

11.19.1 功能描述

该文件中定义了一些访问段寄存器或与段寄存器有关的内存操作函数。

11.19.2 代码注释

列表 linux/include/asm/segment.h 文件

```

1  // 读取 fs 段中指定地址处的字节。
2  // 参数: addr - 指定的内存地址。
3  // %0 - (返回的字节_v); %1 - (内存地址 addr)。
4  // 返回: 返回内存 fs:[addr]处的字节。
5  extern inline unsigned char get_fs_byte(const char * addr)
6  {
7      unsigned register char _v;
8
9      __asm__ ("movb %%fs:%1, %0": "=r" (_v): "m" (*addr));
10     return _v;
11 }
12
13 // 读取 fs 段中指定地址处的字。
14 // 参数: addr - 指定的内存地址。
15 // %0 - (返回的字_v); %1 - (内存地址 addr)。
16 // 返回: 返回内存 fs:[addr]处的字。
17 extern inline unsigned short get_fs_word(const unsigned short *addr)
18 {
19     unsigned short _v;
20
21     __asm__ ("movw %%fs:%1, %0": "=r" (_v): "m" (*addr));
22     return _v;
23 }
24
25 // 读取 fs 段中指定地址处的长字(4 字节)。
26 // 参数: addr - 指定的内存地址。
27 // %0 - (返回的长字_v); %1 - (内存地址 addr)。
28 // 返回: 返回内存 fs:[addr]处的长字。
29 extern inline unsigned long get_fs_long(const unsigned long *addr)
30 {
31     unsigned long _v;
32
33     __asm__ ("movl %%fs:%1, %0": "=r" (_v): "m" (*addr)); \
34     return _v;
35 }
36
37 // 将一字节存放在 fs 段中指定内存地址处。

```



```

// 参数: val - 字节值; addr - 内存地址。
// %0 - 寄存器(字节值 val); %1 - (内存地址 addr)。
25 extern inline void put\_fs\_byte(char val, char *addr)
26 {
27     __asm__ ("movb %0, %%fs:%1":: "r" (val), "m" (*addr));
28 }
29
//// 将一字存放在 fs 段中指定内存地址处。
// 参数: val - 字值; addr - 内存地址。
// %0 - 寄存器(字值 val); %1 - (内存地址 addr)。
30 extern inline void put\_fs\_word(short val, short * addr)
31 {
32     __asm__ ("movw %0, %%fs:%1":: "r" (val), "m" (*addr));
33 }
34
//// 将一长字存放在 fs 段中指定内存地址处。
// 参数: val - 长字值; addr - 内存地址。
// %0 - 寄存器(长字值 val); %1 - (内存地址 addr)。
35 extern inline void put\_fs\_long(unsigned long val, unsigned long * addr)
36 {
37     __asm__ ("movl %0, %%fs:%1":: "r" (val), "m" (*addr));
38 }
39
40 /*
41  * Someone who knows GNU asm better than I should double check the followig.
42  * It seems to work, but I don't know if I'm doing something subtly wrong.
43  * --- TYT, 11/24/91
44  * [ nothing wrong here, Linus ]
45  */
46
/*
 * 比我更懂 GNU 汇编的人应该仔细检查下面的代码。这些代码能使用，但我不知道是否
 * 含有一些小错误。
 * --- TYT, 1991 年 11 月 24 日
 * [ 这些代码没有错误, Linus ]
 */
47
//// 取 fs 段寄存器值(选择符)。
// 返回: fs 段寄存器值。
48 extern inline unsigned long get\_fs()
49 {
50     unsigned short _v;
51     __asm__ ("mov %%fs, %%ax":: "=a" (_v));
52     return _v;
53 }
54
//// 取 ds 段寄存器值。
// 返回: ds 段寄存器值。
55 extern inline unsigned long get\_ds()
56 {
57     unsigned short _v;
58     __asm__ ("mov %%ds, %%ax":: "=a" (_v));
59     return _v;
60 }

```

```

60  // 设置 fs 段寄存器。
61  // 参数: val - 段值 (选择符)。
62  extern inline void set_fs(unsigned long val)
63  {
64      __asm__ ("mov %0, %%fs::%a" ((unsigned short) val));
65  }
66

```

11.20 system.h 文件

11.20.1 功能描述

该文件中定义了设置或修改描述符/中断门等的嵌入式汇编宏。

其中，函数 `move_to_user_mode()` 是用于内核在初始化结束时切换到初始进程（任务 0）。所使用的方法是模拟中断调用返回过程，也即利用指令 `iret` 运行初始任务 0。

在切换到任务 0 之前，首先设置堆栈，模拟具有特权层切换的刚进入中断调用过程时堆栈的内容布置情况，见下图所示。然后执行 `iret` 指令，从而引起系统切换到任务 0 去执行。

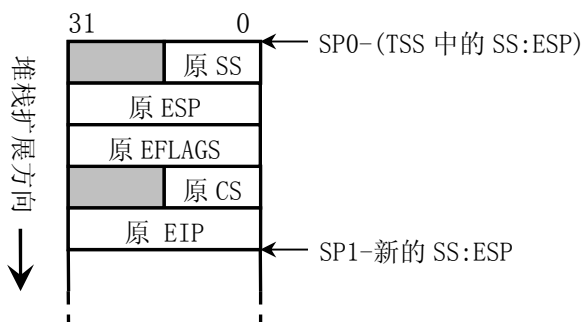


图 中断调用层间切换时堆栈内容

任务 0 是一个特殊进程，它的数据段和代码段直接映射到内核代码和数据空间，即从物理地址 0 开始的 640K 内存空间，其堆栈地址也即内核代码所使用的堆栈。因此图中堆栈中的原 SS 和原 ESP 是直接将有内核的堆栈指针压入堆栈的。

11.20.2 代码注释

列表 linux/include/asm/system.h 文件

```

// 切换到用户模式运行。
// 该函数利用 iret 指令实现从内核模式切换到用户模式（初始任务 0）。
1 #define move_to_user_mode() \
2 __asm__ ("movl %%esp, %%eax\n\t" \           // 保存堆栈指针 esp 到 eax 寄存器中。
3         "pushl $0x17\n\t" \               // 首先将堆栈段选择符(SS)入栈。

```

```

4      "pushl %%eax|n|t" \           // 然后将保存的堆栈指针值(esp)入栈。
5      "pushfl|n|t" \               // 将标志寄存器(eflags)内容入栈。
6      "pushl $0x0f|n|t" \         // 将内核代码段选择符(cs)入栈。
7      "pushl $1f|n|t" \           // 将下面标号 1 的偏移地址(eip)入栈。
8      "iret|n" \                   // 执行中断返回指令, 则会跳转到下面标号 1 处。
9      "l:\tmovl $0x17, %%eax|n|t" \ // 此时开始执行任务 0,
10     "movw %%ax, %%ds|n|t" \       // 初始化段寄存器指向本局部表的数据段。
11     "movw %%ax, %%es|n|t" \
12     "movw %%ax, %%fs|n|t" \
13     "movw %%ax, %%gs" \
14     ::: "ax")
15
16 #define sti() __asm__ ("sti":)    // 开中断嵌入汇编宏函数。
17 #define cli() __asm__ ("cli":)    // 关中断。
18 #define nop() __asm__ ("nop":)    // 空操作。
19
20 #define iret() __asm__ ("iret":)   // 中断返回。
21
22     //// 设置门描述符宏函数。
23     // 参数: gate_addr - 描述符地址; type - 描述符中类型域值; dpl - 描述符特权层值; addr - 偏移地址。
24     // %0 - (由 dpl, type 组合成的类型标志字); %1 - (描述符低 4 字节地址);
25     // %2 - (描述符高 4 字节地址); %3 - edx(程序偏移地址 addr); %4 - eax(高字中含有段选择符)。
26 #define set_gate(gate_addr, type, dpl, addr) \
27     __asm__ ("movw %%dx, %%ax|n|t" \           // 将偏移地址低字与选择符组合成描述符低 4 字节(eax)。
28             "movw %0, %%dx|n|t" \             // 将类型标志字与偏移高字组合成描述符高 4 字节(edx)。
29             "movl %%eax, %1|n|t" \             // 分别设置门描述符的低 4 字节和高 4 字节。
30             "movl %%edx, %2" \
31             : \
32             : "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
33             "o" (*(char *) (gate_addr)), \
34             "o" (*(4+(char *) (gate_addr))), \
35             "d" ((char *) (addr)), "a" (0x00080000))
36
37     //// 设置中断门函数。
38     // 参数: n - 中断号; addr - 中断程序偏移地址。
39     // &idt[n] 对应中断号在中断描述符表中的偏移值; 中断描述符的类型是 14, 特权级是 0。
40 #define set_intr_gate(n, addr) \
41     set_gate(&idt[n], 14, 0, addr)
42
43     //// 设置陷阱门函数。
44     // 参数: n - 中断号; addr - 中断程序偏移地址。
45     // &idt[n] 对应中断号在中断描述符表中的偏移值; 中断描述符的类型是 15, 特权级是 0。
46 #define set_trap_gate(n, addr) \
47     set_gate(&idt[n], 15, 0, addr)
48
49     //// 设置系统调用门函数。
50     // 参数: n - 中断号; addr - 中断程序偏移地址。
51     // &idt[n] 对应中断号在中断描述符表中的偏移值; 中断描述符的类型是 15, 特权级是 3。
52 #define set_system_gate(n, addr) \
53     set_gate(&idt[n], 15, 3, addr)
54
55     //// 设置段描述符函数。
56     // 参数: gate_addr - 描述符地址; type - 描述符中类型域值; dpl - 描述符特权层值;

```











```

// base - 段的基地址; limit - 段限长。(参见段描述符的格式)
42 #define set_seg_desc(gate_addr, type, dpl, base, limit) {\
43     *(gate_addr) = ((base) & 0xff000000) | \           // 描述符低 4 字节。
44                     (((base) & 0x00ff0000)>>16) | \
45                     ((limit) & 0xf0000) | \
46                     ((dpl)<<13) | \
47                     (0x00408000) | \
48                     ((type)<<8); \
49     *((gate_addr)+1) = (((base) & 0x0000ffff)<<16) | \ // 描述符高 4 字节。
50                     ((limit) & 0x0ffff); }
51
///// 在全局表中设置任务状态段/局部表描述符。
// 参数: n - 在全局表中描述符项 n 所对应的地址; addr - 状态段/局部表所在内存的基地址。
//      type - 描述符中的标志类型字节。
// %0 - eax(地址 addr); %1 - (描述符项 n 的地址); %2 - (描述符项 n 的地址偏移 2 处);
// %3 - (描述符项 n 的地址偏移 4 处); %4 - (描述符项 n 的地址偏移 5 处);
// %5 - (描述符项 n 的地址偏移 6 处); %6 - (描述符项 n 的地址偏移 7 处);
52 #define set_tssldt_desc(n, addr, type) \
53 __asm_ ("movw $104, %1\n\t" \           // 将 TSS 长度放入描述符长度域(第 0-1 字节)。
54         "movw %%ax, %2\n\t" \           // 将基地址的低字放入描述符第 2-3 字节。
55         "rorl $16, %%eax\n\t" \         // 将基地址高字移入 ax 中。
56         "movb %%al, %3\n\t" \           // 将基地址高字中低字节移入描述符第 4 字节。
57         "movb $" type ", %4\n\t" \     // 将标志类型字节移入描述符的第 5 字节。
58         "movb $0x00, %5\n\t" \         // 描述符的第 6 字节置 0。
59         "movb %%ah, %6\n\t" \         // 将基地址高字中高字节移入描述符第 7 字节。
60         "rorl $16, %%eax" \             // eax 清零。
61         :: "a" (addr), "m" (*(n)), "m" (*(n+2)), "m" (*(n+4)), \
62         "m" (*(n+5)), "m" (*(n+6)), "m" (*(n+7)) \
63         )
64
///// 在全局表中设置任务状态段描述符。
// n - 是该描述符的指针; addr - 是描述符中的基地址值。任务状态段描述符的类型是 0x89。
65 #define set_tss_desc(n, addr) set_tssldt_desc((char *) (n), addr, "0x89")
///// 在全局表中设置局部表描述符。
// n - 是该描述符的指针; addr - 是描述符中的基地址值。局部表描述符的类型是 0x82。
66 #define set_ldt_desc(n, addr) set_tssldt_desc((char *) (n), addr, "0x82")
67

```

11.21 include/linux/目录下的文件

列表 linux/include/linux/目录下的文件

Name	Size	Last modified (GMT)	Description
 config.h	1289 bytes	1991-12-08 18:37:16	m
 fdreg.h	2466 bytes	1991-11-02 10:48:44	m
 fs.h	5474 bytes	1991-12-01 19:48:26	m
 hdreg.h	1968 bytes	1991-10-13 15:32:15	m
 head.h	304 bytes	1991-06-19 19:24:13	m
 kernel.h	734 bytes	1991-12-02 03:19:07	m
 mm.h	219 bytes	1991-07-29 17:51:12	m
 sched.h	5838 bytes	1991-11-20 14:40:46	m
 sys.h	2588 bytes	1991-11-25 20:15:35	m
 tty.h	2173 bytes	1991-09-21 11:58:05	m

11.22 config.h 文件

11.22.1 功能描述

内核配置头文件。定义使用的键盘语言类型和硬盘类型（HD_TYPE）可选项。

11.22.2 代码注释

列表 linux/include/linux/config.h 文件

```

1 #ifndef CONFIG\_H
2 #define CONFIG\_H
3
4 /*
```

```

5  * The root-device is no longer hard-coded. You can change the default
6  * root-device by changing the line ROOT_DEV = XXX in boot/bootsect.s
7  */
/*
 * 根文件系统设备已不再是硬编码的了。通过修改 boot/bootsect.s 文件中行
 * ROOT_DEV = XXX，你可以改变根设备的默认设置值。
 */

8
9 /*
10 * define your keyboard here -
11 * KBD_FINNISH for Finnish keyboards
12 * KBD_US for US-type
13 * KBD_GR for German keyboards
14 * KBD_FR for Frech keyboard
15 */
/*
 * 在这里定义你的键盘类型 -
 * KBD_FINNISH 是芬兰键盘。
 * KBD_US 是美式键盘。
 * KBD_GR 是德式键盘。
 * KBD_FR 是法式键盘。
 */
16 /*#define KBD_US */
17 /*#define KBD_GR */
18 /*#define KBD_FR */
19 #define KBD_FINNISH
20
21 /*
22 * Normally, Linux can get the drive parameters from the BIOS at
23 * startup, but if this for some unfathomable reason fails, you'd
24 * be left stranded. For this case, you can define HD_TYPE, which
25 * contains all necessary info on your harddisk.
26 *
27 * The HD_TYPE macro should look like this:
28 *
29 * #define HD_TYPE { head, sect, cyl, wpcom, lzone, ctl}
30 *
31 * In case of two harddisks, the info should be sepatated by
32 * commas:
33 *
34 * #define HD_TYPE { h, s, c, wpcom, lz, ctl }, { h, s, c, wpcom, lz, ctl }
35 */
/*
 * 通常，Linux 能够在启动时从 BIOS 中获取驱动器德参数，但是若由于未知原因
 * 而没有得到这些参数时，会使程序束手无策。对于这种情况，你可以定义 HD_TYPE，
 * 其中包括硬盘的所有信息。
 *
 * HD_TYPE 宏应该象下面这样的形式：
 *
 * #define HD_TYPE { head, sect, cyl, wpcom, lzone, ctl}
 *
 * 对于有两个硬盘的情况，参数信息需用逗号分开：
 *

```

```

    * #define HD_TYPE { h, s, c, wpcom, lz, ctl }, {h, s, c, wpcom, lz, ctl }
    */
36 /*
37  This is an example, two drives, first is type 2, second is type 3:
38
39 #define HD_TYPE { 4, 17, 615, 300, 615, 8 }, { 6, 17, 615, 300, 615, 0 }
40
41 NOTE: ctl is 0 for all drives with heads<=8, and ctl=8 for drives
42 with more than 8 heads.
43
44 If you want the BIOS to tell what kind of drive you have, just
45 leave HD_TYPE undefined. This is the normal thing to do.
46 */
    /*
    * 下面是一个例子，两个硬盘，第 1 个是类型 2，第 2 个是类型 3：
    *
    * #define HD_TYPE { 4, 17, 615, 300, 615, 8 }, {6, 17, 615, 300, 615, 0 }
    *
    * 注意：对应所有硬盘，若其磁头数<=8，则 ctl 等于 0，若磁头数多于 8 个，
    * 则 ctl=8。
    *
    * 如果你想让 BIOS 给出硬盘的类型，那么只需不定义 HD_TYPE。这是默认操作。
    */
47
48 #endif
49

```

11.23 fdreg.h 头文件

11.23.1 功能描述

该头文件用以说明软盘系统常用到的一些参数以及所使用的 I/O 端口。由于软盘驱动器的控制比较烦琐，命令也多，因此在阅读代码之前，最好先参考有关微型计算机控制接口原理的书籍，了解软盘控制器(FDC)的工作原理，然后你就会觉得这里的定义还是比较合理有序的。

在编程时需要访问 4 个端口，分别对应一个或多个寄存器。对于 1.2M 的软盘控制器有以下一些端口。

表 软盘控制器端口

I/O 端口	读写性	寄存器名称
0x3f2	只写	数字输出寄存器(数字控制寄存器)
0x3f4	只读	FDC 主状态寄存器
0x3f5	读/写	FDC 数据寄存器
0x3f7	只读	数字输入寄存器
0x3f7	只写	磁盘控制寄存器(传输率控制)

数字输出端口（数字控制端口）是一个 8 位寄存器，它控制驱动器马达开启、驱动器选择、启动/复位 FDC 以及允许/禁止 DMA 及中断请求。

FDC 的主状态寄存器也是一个 8 位寄存器，用于反映软盘控制器 FDC 和软盘驱动器 FDD 的基本状态。通常，在 CPU 向 FDC 发送命令之前或从 FDC 获取操作结果之前，都要读取主状态寄存器的状态位，以判别当前 FDC 数据寄存器是否就绪，以及确定数据传送的方向。

FDC 的数据端口对应多个寄存器（只写型命令寄存器和参数寄存器、只读型结果寄存器），但任一时刻只能有一个寄存器出现在数据端口 0x3f5。在访问只写型寄存器时，主状态控制的 DIO 方向位必须为 0（CPU → FDC），访问只读型寄存器时则反之。在读取结果时只有在 FDC 不忙之后才算读完结果，通常结果数据最多有 7 个字节。

软盘控制器共可以接受 15 条命令。每个命令均经历三个阶段：命令阶段、执行阶段和结果阶段。

命令阶段是 CPU 向 FDC 发送命令字节和参数字节。每条命令的第一个字节总是命令字节（命令码）。其后跟着 0—8 字节的参数。执行阶段是 FDC 执行命令规定的操作。在执行阶段 CPU 是不加干预的，一般是通过 FDC 发出中断请求获知命令执行的结束。如果 CPU 发出的 FDC 命令是传送数据，则 FDC 可以以中断方式或 DMA 方式进行。中断方式每次传送 1 字节。DMA 方式是在 DMA 控制器管理下，FDC 与内存进行数据的传输直至全部数据传送完。此时 DMA 控制器会将传输字节计数终止信号通知 FDC，最后由 FDC 发出中断请求信号告知 CPU 执行阶段结束。结果阶段是由 CPU 读取 FDC 数据寄存器返回值，从而获得 FDC 命令执行的结果。返回结果数据的长度为 0—7 字节。对于没有返回结果数据的命令，则应向 FDC 发送检测中断状态命令获得操作的状态。

11.23.2 文件注释

列表 linux/include/linux/fdreg.h 文件

```

1 /*
2  * This file contains some defines for the floppy disk controller.
3  * Various sources. Mostly "IBM Microcomputers: A Programmers
4  * Handbook", Sanches and Canton.
5  */
6 /*
7  * 该文件中含有一些软盘控制器的一些定义。这些信息有多处来源，大多数取自 Sanches 和 Canton
8  * 编著的"IBM 微型计算机：程序员手册"一书。
9  */
10 #ifndef FDREG_H // 该定义用来排除代码中重复包含此头文件。
11 #define FDREG_H
12
13 // 一些软盘类型函数的原型说明。
14 extern int ticks_to_floppy_on(unsigned int nr);
15 extern void floppy_on(unsigned int nr);
16 extern void floppy_off(unsigned int nr);
17 extern void floppy_select(unsigned int nr);
18 extern void floppy_deselect(unsigned int nr);
19
20 // 下面是有关软盘控制器一些端口和符号的定义。
21 /* Fd controller regs. S&C, about page 340 */
22 /* 软盘控制器(FDC)寄存器端口。摘自 S&C 书中约 340 页 */
23 #define FD_STATUS 0x3f4 // 主状态寄存器端口。
24 #define FD_DATA 0x3f5 // 数据端口。
25 #define FD_DOR 0x3f2 /* Digital Output Register */
26 // 数字输出寄存器（也称为数字控制寄存器）。
27 #define FD_DIR 0x3f7 /* Digital Input Register (read) */
28 // 数字输入寄存器。
29 #define FD_DCR 0x3f7 /* Diskette Control Register (write)*/

```

```

21                                     // 数据传输率控制寄存器。
22 /* Bits of main status register */
23 /* 主状态寄存器各比特位的含义 */
24 #define STATUS_BUSYMASK 0x0F /* drive busy mask */
25                                     // 驱动器忙位（每位对应一个驱动器）。
26 #define STATUS_BUSY 0x10 /* FDC busy */
27                                     // 软盘控制器忙。
28 #define STATUS_DMA 0x20 /* 0- DMA mode */
29                                     // 0 - 为 DMA 数据传输模式，1 - 为非 DMA 模式。
30 #define STATUS_DIR 0x40 /* 0- cpu->fdc */
31                                     // 传输方向：0 - CPU → fdc，1 - 相反。
32 #define STATUS_READY 0x80 /* Data reg ready */
33                                     // 数据寄存器就绪位。
34
35 /* Bits of FD_ST0 */
36 /* 状态字节 0 (ST0) 各比特位的含义 */
37 #define ST0_DS 0x03 /* drive select mask */
38                                     // 驱动器选择号（发生中断时驱动器号）。
39 #define ST0_HA 0x04 /* Head (Address) */
40                                     // 磁头号。
41 #define ST0_NR 0x08 /* Not Ready */
42                                     // 磁盘驱动器未准备好。
43 #define ST0_ECE 0x10 /* Equipment chech error */
44                                     // 设备检测出错（零磁道校准出错）。
45 #define ST0_SE 0x20 /* Seek end */
46                                     // 寻道或重新校正操作执行结束。
47 #define ST0_INTR 0xC0 /* Interrupt code mask */
48                                     // 中断代码位（中断原因），00 - 命令正常结束；
49                                     // 01 - 命令异常结束；10 - 命令无效；11 - FDD 就绪状态改变。
50
51 /* Bits of FD_ST1 */
52 /* 状态字节 1 (ST1) 各比特位的含义 */
53 #define ST1_MAM 0x01 /* Missing Address Mark */
54                                     // 未找到地址标志 (ID AM)。
55 #define ST1_WP 0x02 /* Write Protect */
56                                     // 写保护。
57 #define ST1_ND 0x04 /* No Data - unreadable */
58                                     // 未找到指定的扇区。
59 #define ST1_OR 0x10 /* OverRun */
60                                     // 数据传输超时（DMA 控制器故障）。
61 #define ST1_CRC 0x20 /* CRC error in data or addr */
62                                     // CRC 检验出错。
63 #define ST1_EOC 0x80 /* End Of Cylinder */
64                                     // 访问超过一个磁道上的最大扇区号。
65
66 /* Bits of FD_ST2 */
67 /* 状态字节 2 (ST2) 各比特位的含义 */
68 #define ST2_MAM 0x01 /* Missing Address Mark (again) */
69                                     // 未找到数据地址标志。
70 #define ST2_BC 0x02 /* Bad Cylinder */
71                                     // 磁道坏。
72 #define ST2_SNS 0x04 /* Scan Not Satisfied */

```

```

49 #define ST2_SEH          0x08          // 检索（扫描）条件不满足。
                                         /* Scan Equal Hit */
                                         // 检索条件满足。
50 #define ST2_WC          0x10          /* Wrong Cylinder */
                                         // 磁道（柱面）号不符。
51 #define ST2_CRC          0x20          /* CRC error in data field */
                                         // 数据场 CRC 校验错。
52 #define ST2_CM          0x40          /* Control Mark = deleted */
                                         // 读数据遇到删除标志。

53
54 /* Bits of FD_ST3 */
   /* 状态字节 3 (ST3) 各比特位的含义 */
55 #define ST3_HA          0x04          /* Head (Address) */
                                         // 磁头号。
56 #define ST3_TZ          0x10          /* Track Zero signal (1=track 0) */
                                         // 零磁道信号。
57 #define ST3_WP          0x40          /* Write Protect */
                                         // 写保护。

58
59 /* Values for FD_COMMAND */
   /* 软盘命令码 */
60 #define FD_RECALIBRATE  0x07          /* move to track 0 */
                                         // 重新校正(磁头退到零磁道)。
61 #define FD_SEEK        0x0F          /* seek track */
                                         // 磁头寻道。
62 #define FD_READ        0xE6          /* read with MT, MFM, SKip deleted */
                                         // 读数据 (MT 多磁道操作, MFM 格式, 跳过删除数据)。
63 #define FD_WRITE       0xC5          /* write with MT, MFM */
                                         // 写数据 (MT, MFM)。
64 #define FD_SENSEI      0x08          /* Sense Interrupt Status */
                                         // 检测中断状态。
65 #define FD_SPECIFY     0x03          /* specify HUT etc */
                                         // 设定驱动器参数 (步进速率、磁头卸载时间等)。

66
67 /* DMA commands */
   /* DMA 命令 */
68 #define DMA_READ        0x46          // DMA 读盘, DMA 方式字 (送 DMA 端口 12, 11)。
69 #define DMA_WRITE      0x4A          // DMA 写盘, DMA 方式字。
70
71 #endif
72

```

11.24 fs.h 文件

11.24.1 功能描述

11.24.2 代码注释

列表 linux/include/linux/fs.h 文件

```

1  /*
2  * This file has definitions for some important file table
3  * structures etc.
4  */
5  /*
6  * 本文件含有某些重要文件表结构的定义等。
7  */
8
9  #ifndef FS\_H
10 #define FS\_H
11
12 #include <sys/types.h>    // 类型头文件。定义了基本的系统数据类型。
13
14 /* devices are as follows: (same as minix, so we can use the minix
15 * file system. These are major numbers.)
16 *
17 * 0 - unused (nodev)
18 * 1 - /dev/mem
19 * 2 - /dev/fd
20 * 3 - /dev/hd
21 * 4 - /dev/ttyx
22 * 5 - /dev/tty
23 * 6 - /dev/lp
24 * 7 - unnamed pipes
25 */
26 /*
27 * 系统所含的设备如下：（与 minix 系统的一样，所以我们可以使用 minix 的
28 * 文件系统。以下这些是主设备号。）
29 *
30 * 0 - 没有用到（nodev）
31 * 1 - /dev/mem      内存设备。
32 * 2 - /dev/fd       软盘设备。
33 * 3 - /dev/hd       硬盘设备。
34 * 4 - /dev/ttyx     tty 串行终端设备。
35 * 5 - /dev/tty      tty 终端设备。
36 * 6 - /dev/lp       打印设备。
37 * 7 - unnamed pipes 没有命名的管道。
38 */
39
40 #define IS\_SEEKABLE(x) ((x)>=1 && (x)<=3)    // 是否是寻找定位的设备。
41
42 #define READ 0
43 #define WRITE 1
44 #define READA 2    /* read-ahead - don't pause */
45 #define WRITEA 3    /* "write-ahead" - silly, but somewhat useful */
46
47 void buffer\_init(long buffer_end);
48
49 #define MAJOR(a) (((unsigned)(a))>>8)    // 取高字节（主设备号）。
50 #define MINOR(a) ((a)&0xff)            // 取低字节（次设备号）。

```

```

35
36 #define NAME_LEN 14 // 名字长度值。
37 #define ROOT_INO 1 // 根 i 节点。
38
39 #define I_MAP_SLOTS 8 // i 节点位图槽数。
40 #define Z_MAP_SLOTS 8 // 逻辑块（区段块）位图槽数。
41 #define SUPER_MAGIC 0x137F // 文件系统魔数。
42
43 #define NR_OPEN 20 // 打开文件数。
44 #define NR_INODE 32
45 #define NR_FILE 64
46 #define NR_SUPER 8
47 #define NR_HASH 307
48 #define NR_BUFFERS nr_buffers
49 #define BLOCK_SIZE 1024 // 数据块长度。
50 #define BLOCK_SIZE_BITS 10 // 数据块长度所占比特位数。
51 #ifndef NULL
52 #define NULL ((void *) 0)
53 #endif
54
// 每个逻辑块可存放的 i 节点数。
55 #define INODES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct d_inode)))
// 每个逻辑块可存放的目录项数。
56 #define DIR_ENTRIES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct dir_entry)))
57
// 管道头、管道尾、管道大小、管道空?、管道满?、管道头指针递增。
58 #define PIPE_HEAD(inode) ((inode).i_zone[0])
59 #define PIPE_TAIL(inode) ((inode).i_zone[1])
60 #define PIPE_SIZE(inode) ((PIPE_HEAD(inode)-PIPE_TAIL(inode))&(PAGE_SIZE-1))
61 #define PIPE_EMPTY(inode) (PIPE_HEAD(inode)==PIPE_TAIL(inode))
62 #define PIPE_FULL(inode) (PIPE_SIZE(inode)==(PAGE_SIZE-1))
63 #define INC_PIPE(head) \
64 __asm__("incl %0\n\tandl $4095, %0": "m" (head))
65
66 typedef char buffer_block[BLOCK_SIZE]; // 块缓冲区。
67
// 缓冲区头数据结构。（极为重要!!!）
// 在程序中常用 bh 来表示 buffer_head 类型的缩写。
68 struct buffer_head {
69     char * b_data; // pointer to data block (1024 bytes) // 数据块。
70     unsigned long b_blocknr; // block number // 块号。
71     unsigned short b_dev; // device (0 = free) // 数据源的设备号。
72     unsigned char b_uptodate; // 更新标志：表示数据是否已更新。
73     unsigned char b_dirty; // 0-clean, 1-dirty // 修改标志：0-未修改，1-已修
改。
74     unsigned char b_count; // users using this block // 使用的用户数。
75     unsigned char b_lock; // 0 - ok, 1 -locked // 缓冲区是否被锁定。
76     struct task_struct * b_wait; // 指向等待该缓冲区解锁的任务。
77     struct buffer_head * b_prev; // 前一块（这四个指针用于缓冲区的管理）。
78     struct buffer_head * b_next; // 下一块。
79     struct buffer_head * b_prev_free; // 前一空闲块。
80     struct buffer_head * b_next_free; // 下一空闲块。
81 };

```

```

82 // 磁盘上的索引节点(i 节点)数据结构。
83 struct d_inode {
84     unsigned short i_mode;        // 文件类型和属性(rwx 位)。
85     unsigned short i_uid;        // 用户 id (文件拥有者标识符)。
86     unsigned long i_size;        // 文件大小 (字节数)。
87     unsigned long i_time;        // 修改时间 (自 1970.1.1:0 算起, 秒)。
88     unsigned char i_gid;        // 组 id (文件拥有者所在的组)。
89     unsigned char i_nlinks;      // 链接数 (多少个文件目录项指向该 i 节点)。
90     unsigned short i_zone[9];    // 直接 (0-6)、间接 (7) 或双重间接 (8) 逻辑块号。
                                   // zone 是区的意思, 可译成区段, 或逻辑块。
91 };
92 // 这是在内存中的 i 节点结构。前 7 项与 d_inode 完全一样。
93 struct m_inode {
94     unsigned short i_mode;        // 文件类型和属性(rwx 位)。
95     unsigned short i_uid;        // 用户 id (文件拥有者标识符)。
96     unsigned long i_size;        // 文件大小 (字节数)。
97     unsigned long i_mtime;      // 修改时间 (自 1970.1.1:0 算起, 秒)。
98     unsigned char i_gid;        // 组 id (文件拥有者所在的组)。
99     unsigned char i_nlinks;      // 文件目录项链接数。
100    unsigned short i_zone[9];    // 直接 (0-6)、间接 (7) 或双重间接 (8) 逻辑块号。
101    /* these are in memory also */
102    struct task_struct * i_wait;  // 等待该 i 节点的进程。
103    unsigned long i_atime;        // 最后访问时间。
104    unsigned long i_ctime;        // i 节点自身修改时间。
105    unsigned short i_dev;        // i 节点所在的设备号。
106    unsigned short i_num;        // i 节点号。
107    unsigned short i_count;      // i 节点被使用的次数, 0 表示该 i 节点空闲。
108    unsigned char i_lock;        // 锁定标志。
109    unsigned char i_dirt;        // 已修改(脏)标志。
110    unsigned char i_pipe;        // 管道标志。
111    unsigned char i_mount;       // 安装标志。
112    unsigned char i_seek;        // 搜寻标志(lseek 时)。
113    unsigned char i_update;      // 更新标志。
114 };
115 // 文件结构 (用于在文件句柄与 i 节点之间建立关系)
116 struct file {
117     unsigned short f_mode;      // 文件操作模式 (RW 位)
118     unsigned short f_flags;     // 文件打开和控制的标志。
119     unsigned short f_count;     // 对应文件句柄 (文件描述符) 数。
120     struct m_inode * f_inode;   // 指向对应 i 节点。
121     off_t f_pos;               // 文件位置 (读写偏移值)。
122 };
123 // 内存中磁盘超级块结构。
124 struct super_block {
125     unsigned short s_ninodes;   // 节点数。
126     unsigned short s_nzones;    // 逻辑块数。
127     unsigned short s_imap_blocks; // i 节点位图所占用的数据块数。
128     unsigned short s_zmap_blocks; // 逻辑块位图所占用的数据块数。
129     unsigned short s_firstdatazone; // 第一个数据逻辑块号。

```

```

130     unsigned short s_log_zone_size; // log(数据块数/逻辑块)。(以 2 为底)。
131     unsigned long s_max_size;      // 文件最大长度。
132     unsigned short s_magic;        // 文件系统魔数。
133     /* These are only in memory */
134     struct buffer head * s_imap[8]; // i 节点位图缓冲块指针数组(占用 8 块, 可表示 64M)。
135     struct buffer head * s_zmap[8]; // 逻辑块位图缓冲块指针数组(占用 8 块)。
136     unsigned short s_dev;          // 超级块所在的设备号。
137     struct m\_inode * s_isup;        // 被安装的文件系统根目录的 i 节点。(isup=super i)
138     struct m\_inode * s_imount;      // 被安装到的 i 节点。
139     unsigned long s_time;          // 修改时间。
140     struct task\_struct * s_wait;    // 等待该超级块的进程。
141     unsigned char s_lock;          // 被锁定标志。
142     unsigned char s_rd_only;       // 只读标志。
143     unsigned char s_dirt;          // 已修改(脏)标志。
144 };
145
146 // 磁盘上超级块结构。上面 125-132 行完全一样。
147 struct d\_super\_block {
148     unsigned short s_ninodes;      // 节点数。
149     unsigned short s_nzones;       // 逻辑块数。
150     unsigned short s_imap_blocks;  // i 节点位图所占用的数据块数。
151     unsigned short s_zmap_blocks;  // 逻辑块位图所占用的数据块数。
152     unsigned short s_firstdatazone; // 第一个数据逻辑块。
153     unsigned short s_log_zone_size; // log(数据块数/逻辑块)。(以 2 为底)。
154     unsigned long s_max_size;      // 文件最大长度。
155     unsigned short s_magic;        // 文件系统魔数。
156 };
157
158 // 文件目录项结构。
159 struct dir\_entry {
160     unsigned short inode;          // i 节点。
161     char name[NAME\_LEN];           // 文件名。
162 };
163
164 extern struct m\_inode inode\_table[NR\_INODE]; // 定义 i 节点表数组(32 项)。
165 extern struct file file\_table[NR\_FILE];     // 文件表数组(64 项)。
166 extern struct super\_block super\_block[NR\_SUPER]; // 超级块数组(8 项)。
167 extern struct buffer head * start\_buffer;    // 缓冲区起始内存位置。
168 extern int nr\_buffers;                     // 缓冲块数。
169
170 // 磁盘操作函数原型。
171 // 检测驱动器中软盘是否改变。
172 extern void check\_disk\_change(int dev);
173 // 检测指定软驱中软盘更换情况。如果软盘更换了则返回 1, 否则返回 0。
174 extern int floppy\_change(unsigned int nr);
175 // 设置启动指定驱动器所需等待的时间(设置等待定时器)。
176 extern int ticks\_to\_floppy\_on(unsigned int dev);
177 // 启动指定驱动器。
178 extern void floppy\_on(unsigned int dev);
179 // 关闭指定的软盘驱动器。
180 extern void floppy\_off(unsigned int dev);
181
182 // 以下是文件系统操作管理用的函数原型。

```



```

    // 将 i 节点指定的文件截为 0。
173 extern void truncate(struct m\_inode * inode);
    // 刷新 i 节点信息。
174 extern void sync\_inodes(void);
    // 等待指定的 i 节点。
175 extern void wait\_on(struct m\_inode * inode);
    // 逻辑块(区段, 磁盘块)位图操作。取数据块 block 在设备上对应的逻辑块号。
176 extern int bmap(struct m\_inode * inode, int block);
    // 创建数据块 block 在设备上对应的逻辑块, 并返回在设备上的逻辑块号。
177 extern int create\_block(struct m\_inode * inode, int block);
    // 获取指定路径名的 i 节点号。
178 extern struct m\_inode * namei(const char * pathname);
    // 根据路径名为打开文件操作作准备。
179 extern int open\_namei(const char * pathname, int flag, int mode,
180     struct m\_inode ** res_inode);
    // 释放一个 i 节点(回写入设备)。
181 extern void iput(struct m\_inode * inode);
    // 从设备读取指定节点号的一个 i 节点。
182 extern struct m\_inode * iget(int dev, int nr);
    // 从 i 节点表(inode_table)中获取一个空闲 i 节点项。
183 extern struct m\_inode * get\_empty\_inode(void);
    // 获取(申请)管道节点。返回为 i 节点指针(如果是 NULL 则失败)。
184 extern struct m\_inode * get\_pipe\_inode(void);
    // 在哈希表中查找指定的数据块。返回找到块的缓冲头指针。
185 extern struct buffer\_head * get\_hash\_table(int dev, int block);
    // 从设备读取指定块(首先会在 hash 表中查找)。
186 extern struct buffer\_head * getblk(int dev, int block);
    // 读/写数据块。
187 extern void ll\_rw\_block(int rw, struct buffer\_head * bh);
    // 释放指定缓冲块。
188 extern void brelse(struct buffer\_head * buf);
    // 读取指定的数据块。
189 extern struct buffer\_head * bread(int dev, int block);
    // 读 4 块缓冲区到指定地址的内存中。
190 extern void bread\_page(unsigned long addr, int dev, int b[4]);
    // 读取头一个指定的数据块, 并标记后续将要读的块。
191 extern struct buffer\_head * breada(int dev, int block, ...);
    // 向设备 dev 申请一个磁盘块(区段, 逻辑块)。返回逻辑块号
192 extern int new\_block(int dev);
    // 释放设备数据区中的逻辑块(区段, 磁盘块)block。复位指定逻辑块 block 的逻辑块位图比特位。
193 extern void free\_block(int dev, int block);
    // 为设备 dev 建立一个新 i 节点, 返回 i 节点号。
194 extern struct m\_inode * new\_inode(int dev);
    // 释放一个 i 节点(删除文件时)。
195 extern void free\_inode(struct m\_inode * inode);
    // 刷新指定设备缓冲区。
196 extern int sync\_dev(int dev);
    // 读取指定设备的超级块。
197 extern struct super\_block * get\_super(int dev);
198 extern int ROOT\_DEV;
199
    // 安装根文件系统。
200 extern void mount\_root(void);

```

201
202 #endif
203

11.25 hdreg.h 文件

11.25.1 功能描述

11.25.2 代码注释

列表 linux/include/linux/hdreg.h 文件

```

1  /*
2  * This file contains some defines for the AT-hd-controller.
3  * Various sources. Check out some definitions (see comments with
4  * a ques).
5  */
/*
* 本文件含有一些 AT 硬盘控制器的定义。来自各种资料。请查证某些
* 定义（带有问号的注释）。
*/
6 #ifndef HDREG_H
7 #define HDREG_H
8
9 /* Hd controller regs. Ref: IBM AT Bios-listing */
/* 硬盘控制器寄存器端口。参见：IBM AT Bios 程序 */
10 #define HD_DATA 0x1f0 /* _CTL when writing */
11 #define HD_ERROR 0x1f1 /* see err-bits */
12 #define HD_NSECTOR 0x1f2 /* nr of sectors to read/write */
13 #define HD_SECTOR 0x1f3 /* starting sector */
14 #define HD_LCYL 0x1f4 /* starting cylinder */
15 #define HD_HCYL 0x1f5 /* high byte of starting cyl */
16 #define HD_CURRENT 0x1f6 /* 101dhhhh, d=drive, hhhh=head */
17 #define HD_STATUS 0x1f7 /* see status-bits */
18 #define HD_PRECOMP HD_ERROR /* same io address, read=error, write=precomp */
19 #define HD_COMMAND HD_STATUS /* same io address, read=status, write=cmd */
20
21 #define HD_CMD 0x3f6 // 控制寄存器端口。
22
23 /* Bits of HD_STATUS */
/* 硬盘状态寄存器各位的定义 (HD_STATUS) */
24 #define ERR_STAT 0x01 // 命令执行错误。
25 #define INDEX_STAT 0x02 // 收到索引。
26 #define ECC_STAT 0x04 /* Corrected error */ // ECC 校验错。
27 #define DRQ_STAT 0x08 // 请求服务。
28 #define SEEK_STAT 0x10 // 寻道结束。
29 #define WRERR_STAT 0x20 // 驱动器故障。
30 #define READY_STAT 0x40 // 驱动器准备好（就绪）。
31 #define BUSY_STAT 0x80 // 控制器忙碌。

```

```

32
33 /* Values for HD_COMMAND */
34 /* 硬盘命令值 (HD_CMD) */
35 #define WIN_RESTORE          0x10    // 驱动器重新校正 (驱动器复位)。
36 #define WIN_READ             0x20    // 读扇区。
37 #define WIN_WRITE            0x30    // 写扇区。
38 #define WIN_VERIFY           0x40    // 扇区检验。
39 #define WIN_FORMAT            0x50    // 格式化磁道。
40 #define WIN_INIT              0x60    // 控制器初始化。
41 #define WIN_SEEK              0x70    // 寻道。
42 #define WIN_DIAGNOSE          0x90    // 控制器诊断。
43 #define WIN_SPECIFY           0x91    // 建立驱动器参数。
44
45 /* Bits for HD_ERROR */
46 /* 错误寄存器各比特位的含义 (HD_ERROR) */
47 // 执行控制器诊断命令时含义与其它命令时的不同。下面分别列出:
48 // =====
49 //          诊断命令时          其它命令时
50 // -----
51 // 0x01      无错误              数据标志丢失
52 // 0x02      控制器出错          磁道 0 错
53 // 0x03      扇区缓冲区错
54 // 0x04      ECC 部件错          命令放弃
55 // 0x05      控制处理器错
56 // 0x10
57 // 0x40
58 // 0x80
59 // ID 未找到
60 // ECC 错误
61 // 坏扇区
62 // -----
63 #define MARK_ERR             0x01    /* Bad address mark ? */
64 #define TRK0_ERR              0x02    /* couldn't find track 0 */
65 #define ABRT_ERR              0x04    /* ? */
66 #define ID_ERR                0x10    /* ? */
67 #define ECC_ERR               0x40    /* ? */
68 #define BBD_ERR               0x80    /* ? */
69
70 // 硬盘分区表结构。参见下面列表后信息。
71 struct partition {
72     unsigned char boot_ind;    /* 0x80 - active (unused) */
73     unsigned char head;        /* ? */
74     unsigned char sector;      /* ? */
75     unsigned char cyl;         /* ? */
76     unsigned char sys_ind;     /* ? */
77     unsigned char end_head;    /* ? */
78     unsigned char end_sector;  /* ? */
79     unsigned char end_cyl;     /* ? */
80     unsigned int start_sect;   /* starting sector counting from 0 */
81     unsigned int nr_sects;     /* nr of sectors in partition */
82 };
83 #endif
84

```

11.25.3 其它信息

硬盘分区表

为了实现多个操作系统共享硬盘资源，硬盘可以在逻辑上分为 1--4 个分区。每个分区之间的扇区号是邻接的。分区表由 4 个表项组成，每个表项由 16 字节组成，对应一个分区的信息，存放有分区的大小和起止的柱面号、磁道号和扇区号，见下表所示。分区表存放在硬盘的 0 柱面 0 头第 1 个扇区的 0x1BE--0x1FD 处。

表 硬盘分区表结构

位置	名称	大小	说明
0x00	boot_ind	字节	引导标志。4 个分区中同时只能有一个分区是可引导的。0x00-不从该分区引导操作系统；0x80-从该分区引导操作系统。
0x01	head	字节	分区起始磁头号。
0x02	sector	字节	分区起始扇区号(位 0-5)和起始柱面号高 2 位(位 6-7)。
0x03	cyl	字节	分区起始柱面号低 8 位。
0x04	sys_ind	字节	分区类型字节。0x0b-DOS；0x80-Old Minix；0x83-Linux ...
0x05	end_head	字节	分区的结束磁头号。
0x06	end_sector	字节	结束扇区号(位 0-5)和结束柱面号高 2 位(位 6-7)。
0x07	end_cyl	字节	结束柱面号低 8 位。
0x08--0x0b	start_sect	长字	分区起始物理扇区号。
0x0c--0x0f	nr_sects	长字	分区占用的扇区数。

11.26 head.h 文件

11.26.1 功能描述

head 头文件，定义了 Intel CPU 中描述符的简单结构，和指定描述符的项号。

11.26.2 代码注释

列表 linux/include/linux/head.h 文件

```
1 #ifndef HEAD\_H
2 #define HEAD\_H
3
4 typedef struct desc\_struct {           // 定义了段描述符的数据结构。该结构仅说明每个描述
5     unsigned long a,b;                 // 符是由 8 个字节构成，每个描述符表共有 256 项。
6 } desc\_table[256];
7
8 extern unsigned long pg\_dir[1024];    // 内存页目录数组。每个目录项为 4 字节。从物理地址 0 开始。
9 extern desc\_table idt,gdt;             // 中断描述符表，全局描述符表。
10
11 #define GDT\_NUL 0                     // 全局描述符表的第 0 项，不用。
```

```

12 #define GDT_CODE 1 // 第 1 项，是内核代码段描述符项。
13 #define GDT_DATA 2 // 第 2 项，是内核数据段描述符项。
14 #define GDT_TMP 3 // 第 3 项，系统段描述符，Linux 没有使用。
15
16 #define LDT_NUL 0 // 每个局部描述符表的第 0 项，不用。
17 #define LDT_CODE 1 // 第 1 项，是用户程序代码段描述符项。
18 #define LDT_DATA 2 // 第 2 项，是用户程序数据段描述符项。
19
20 #endif
21

```

11.27 kernel.h 文件

11.27.1 功能描述

定义了一些内核常用的函数原型等。

11.27.2 代码注释

列表 linux/include/linux/kernel.h

```

1 /*
2  * 'kernel.h' contains some often-used function prototypes etc
3  */
4 /*
5  * 'kernel.h' 定义了一些常用函数的原型等。
6  */
7 // 验证给定地址开始的内存块是否超限。若超限则追加内存。( kernel/fork.c, 24 )。
8 void verify_area(void * addr, int count);
9 // 显示内核出错信息，然后进入死循环。( kernel/panic.c, 16 )。
10 volatile void panic(const char * str);
11 // 标准打印（显示）函数。( init/main.c, 151)。
12 int printf(const char * fmt, ...);
13 // 内核专用的打印信息函数，功能与 printf() 相同。( kernel/printk.c, 21 )。
14 int printk(const char * fmt, ...);
15 // 往 tty 上写指定长度的字符串。( kernel/chr_drv/tty_io.c, 290 )。
16 int tty_write(unsigned ch, char * buf, int count);
17 // 通用内核内存分配函数。( lib/malloc.c, 117)。
18 void * malloc(unsigned int size);
19 // 释放指定对象占用的内存。( lib/malloc.c, 182)。
20 void free_s(void * obj, int size);
21
22 #define free(x) free_s((x), 0)
23
24 /*
25  * This is defined as a macro, but at some point this might become a
26  * real subroutine that sets a flag if it returns true (to do
27  * BSD-style accounting where the process is flagged if it uses root
28  * privs). The implication of this is that you should do normal
29  * permissions checks first, and check suser() last.
30  */

```

```

20 */
/*
 * 下面函数是以宏的形式定义的，但是在某方面来看它可以成为一个真正的子程序，
 * 如果返回是 true 时它将设置标志（如果使用 root 用户权限的进程设置了标志，则用
 * 于执行 BSD 方式的计帐处理）。这意味着你应该首先执行常规权限检查，最后再
 * 检测 suser()。
 */
21 #define suser() (current->euid == 0)           // 检测是否是超级用户。
22
23

```

11.28 mm.h 文件

11.28.1 功能描述

mm.h 是内存管理头文件。其中主要定义了内存页面的大小和几个页面释放函数原型。

11.28.2 代码注释

列表 linux/include/linux/mm.h 文件

```

1 #ifndef MM_H
2 #define MM_H
3
4 #define PAGE_SIZE 4096           // 定义内存页面的大小(字节数)。
5
6 // 取空闲页面函数。返回页面地址。扫描页面映射数组 mem_map[]取空闲页面。
6 extern unsigned long get_free_page(void);
// 在指定物理地址处放置一页。在页目录和页表中放置指定页面信息。
7 extern unsigned long put_page(unsigned long page,unsigned long address);
// 释放物理地址 addr 开始的一页面内存。修改页面映射数组 mem_map[]中引用次数信息。
8 extern void free_page(unsigned long addr);
9
10 #endif
11

```

11.29 sched.h 文件

11.29.1 功能描述

调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。

11.29.2 代码注释

列表 linux/include/linux/sched.h 文件

```

1 #ifndef SCHED_H

```

```

2 #define _SCHED_H
3
4 #define NR_TASKS 64 // 系统中同时最多任务（进程）数。
5 #define HZ 100 // 定义系统时钟滴答频率(1 百赫兹，每个滴答 10ms)
6
7 #define FIRST_TASK task[0] // 任务 0 比较特殊，所以特意给它单独定义一个符号。
8 #define LAST_TASK task[NR_TASKS-1] // 任务数组中的最后一项任务。
9
10 #include <linux/head.h> // head 头文件，定义了段描述符的简单结构，和几个选择符常量。
11 #include <linux/fs.h> // 文件系统头文件。定义文件表结构（file,buffer_head,m_inode 等）。
12 #include <linux/mm.h> // 内存管理头文件。含有页面大小定义和一些页面释放函数原型。
13 #include <signal.h> // 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。
14
15 #if (NR_OPEN > 32)
16 #error "Currently the close-on-exec-flags are in one word, max 32 files/proc"
17 #endif
18
19 // 这里定义了进程运行可能处的状态。
20 #define TASK_RUNNING 0 // 进程正在运行或已准备就绪。
21 #define TASK_INTERRUPTIBLE 1 // 进程处于可中断等待状态。
22 #define TASK_UNINTERRUPTIBLE 2 // 进程处于不可中断等待状态，主要用于 I/O 操作等待。
23 #define TASK_ZOMBIE 3 // 进程处于僵死状态，已经停止运行，但父进程还没发信号。
24 #define TASK_STOPPED 4 // 进程已停止。
25
26 #ifndef NULL
27 #define NULL ((void *) 0) // 定义 NULL 为空指针。
28 #endif
29
30 // 复制进程的页目录页表。Linus 认为这是内核中最复杂的函数之一。（mm/memory.c, 105）
31 extern int copy_page_tables(unsigned long from, unsigned long to, long size);
32 // 释放页表所指定的内存块及页表本身。（mm/memory.c, 150）
33 extern int free_page_tables(unsigned long from, unsigned long size);
34
35 // 调度程序的初始化函数。（kernel/sched.c, 385）
36 extern void sched_init(void);
37 // 进程调度函数。（kernel/sched.c, 104）
38 extern void schedule(void);
39 // 异常(陷阱)中断处理初始化函数，设置中断调用门并允许中断请求信号。（kernel/traps.c, 181）
40 extern void trap_init(void);
41 // 显示内核出错信息，然后进入死循环。（kernel/panic.c, 16）。
42 extern void panic(const char *str);
43 // 往 tty 上写指定长度的字符串。（kernel/chr_drv/tty_io.c, 290）。
44 extern int tty_write(unsigned minor, char *buf, int count);
45
46 typedef int (*fn_ptr)(); // 定义函数指针类型。
47
48 // 下面是数学协处理器使用的结构，主要用于保存进程切换时 i387 的执行状态信息。
49 struct i387_struct {
50     long cwd; // 控制字(Control word)。
51     long swd; // 状态字(Status word)。
52     long twd; // 标记字(Tag word)。
53     long fip; // 协处理器代码指针。
54     long fcs; // 协处理器代码段寄存器。

```



```

46     long    foo;
47     long    fos;
48     long    st_space[20];    /* 8*10 bytes for each FP-reg = 80 bytes */
49 };
50
// 任务状态段数据结构（参见列表后的信息）。
51 struct tss_struct {
52     long    back_link;    /* 16 high bits zero */
53     long    esp0;
54     long    ss0;    /* 16 high bits zero */
55     long    esp1;
56     long    ss1;    /* 16 high bits zero */
57     long    esp2;
58     long    ss2;    /* 16 high bits zero */
59     long    cr3;
60     long    eip;
61     long    eflags;
62     long    eax, ecx, edx, ebx;
63     long    esp;
64     long    ebp;
65     long    esi;
66     long    edi;
67     long    es;    /* 16 high bits zero */
68     long    cs;    /* 16 high bits zero */
69     long    ss;    /* 16 high bits zero */
70     long    ds;    /* 16 high bits zero */
71     long    fs;    /* 16 high bits zero */
72     long    gs;    /* 16 high bits zero */
73     long    ldt;    /* 16 high bits zero */
74     long    trace_bitmap;    /* bits: trace 0, bitmap 16-31 */
75     struct i387_struct i387;
76 };
77
// 这里是任务（进程）数据结构，或称为进程描述符。
// =====
// long state    任务的运行状态（-1 不可运行，0 可运行（就绪），>0 已停止）。
// long counter    任务运行时间计数（递减）（滴答数），运行时间片。
// long priority    运行优先数。任务开始运行时 counter = priority，越大运行越长。
// long signal    信号。是位图，每个比特位代表一种信号，信号值=位偏移值+1。
// struct sigaction sigaction[32] 信号执行属性结构，对应信号将要执行的操作和标志信息。
// long blocked    进程信号屏蔽码（对应信号位图）。
// -----
// int exit_code    任务执行停止的退出码，其父进程会取。
// unsigned long start_code    代码段地址。
// unsigned long end_code    代码长度（字节数）。
// unsigned long end_data    数据长度（字节数）。
// unsigned long brk    总长度（字节数）。
// unsigned long start_stack    堆栈段地址。
// long pid    进程标识号（进程号）。
// long father    父进程号。
// long pgrp    父进程组号。
// long session    会话号。
// long leader    会话首领。

```

```

// unsigned short uid      用户标识号（用户 id）。
// unsigned short euid     有效用户 id。
// unsigned short suid     保存的用户 id。
// unsigned short gid      组标识号（组 id）。
// unsigned short egid     有效组 id。
// unsigned short sgid     保存的组 id。
// long alarm              报警定时值（滴答数）。
// long utime              用户态运行时间（滴答数）。
// long stime              系统态运行时间（滴答数）。
// long cutime             子进程用户态运行时间。
// long cstime             子进程系统态运行时间。
// long start_time         进程开始运行时刻。
// unsigned short used_math 标志：是否使用了协处理器。
// -----
// int tty                  进程使用 tty 的子设备号。-1 表示没有使用。
// unsigned short umask     文件创建属性屏蔽位。
// struct m_inode * pwd     当前工作目录 i 节点结构。
// struct m_inode * root    根目录 i 节点结构。
// struct m_inode * executable 执行文件 i 节点结构。
// unsigned long close_on_exec 执行时关闭文件句柄位图标志。（参见 include/fcntl.h）
// struct file * filp[NR_OPEN] 进程使用的文件表结构。
// -----
// struct desc_struct ldt[3] 本任务的局部表描述符。0-空，1-代码段 cs，2-数据和堆栈段 ds&ss。
// -----
// struct tss_struct tss    本进程的任务状态段信息结构。
// =====
78 struct task_struct {
79 /* these are hardcoded - don't touch */
80     long state; /* -1 unrunnable, 0 runnable, >0 stopped */
81     long counter;
82     long priority;
83     long signal;
84     struct sigaction sigaction[32];
85     long blocked; /* bitmap of masked signals */
86 /* various fields */
87     int exit_code;
88     unsigned long start_code, end_code, end_data, brk, start_stack;
89     long pid, father, pgrp, session, leader;
90     unsigned short uid, euid, suid;
91     unsigned short gid, egid, sgid;
92     long alarm;
93     long utime, stime, cutime, cstime, start_time;
94     unsigned short used_math;
95 /* file system info */
96     int tty; /* -1 if no tty, so it must be signed */
97     unsigned short umask;
98     struct m_inode * pwd;
99     struct m_inode * root;
100    struct m_inode * executable;
101    unsigned long close_on_exec;
102    struct file * filp[NR_OPEN];
103 /* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
104    struct desc_struct ldt[3];

```

```

105 /* tss for this task */
106     struct tss\_struct tss;
107 };
108
109 /*
110  * INIT_TASK is used to set up the first task table, touch at
111  * your own risk!. Base=0, limit=0x9ffff (=640kB)
112  */
113 /*
114  * INIT_TASK 用于设置第 1 个任务表，若想修改，责任自负☺！
115  * 基址 Base = 0，段长 limit = 0x9ffff (=640kB)。
116  */
117 // 对应上面任务结构的第 1 个任务的信息。
118 #define INIT\_TASK \
119 /* state etc */ { 0, 15, 15, \      // state, counter, priority
120 /* signals */ 0, {}, \      // signal, sigaction[32], blocked
121 /* ec, brk... */ 0, 0, 0, 0, 0, \  // exit_code, start_code, end_code, end_data, brk, start_stack
122 /* pid etc. */ 0, -1, 0, 0, 0, \    // pid, father, pgrp, session, leader
123 /* uid etc */ 0, 0, 0, 0, 0, \    // uid, euid, suid, gid, egid, sgid
124 /* alarm */ 0, 0, 0, 0, 0, \    // alarm, utime, stime, cutime, cstime, start_time
125 /* math */ 0, \                // used_math
126 /* fs info */ -1, 0022, NULL, NULL, NULL, 0, \ // tty, umask, pwd, root, executable, close_on_exec
127 /* filp */ {NULL,}, \          // filp[20]
128 { \                            // ldt[3]
129     {0, 0}, \
130     {0x9f, 0xc0fa00}, \ // 代码长 640K, 基址 0x0, G=1, D=1, DPL=3, P=1 TYPE=0x0a
131     {0x9f, 0xc0f200}, \ // 数据长 640K, 基址 0x0, G=1, D=1, DPL=3, P=1 TYPE=0x02
132 }, \
133 /* ldt */ {0, PAGE\_SIZE+(long)&init\_task, 0x10, 0, 0, 0, 0, (long)&pg\_dir, \ // tss
134     0, 0, 0, 0, 0, 0, 0, \
135     0, 0, 0x17, 0x17, 0x17, 0x17, 0x17, 0x17, \
136     LDT(0), 0x80000000, \
137     {} \
138 }, \
139 }
140
141 extern struct task\_struct *task[NR\_TASKS]; // 任务数组。
142 extern struct task\_struct *last\_task\_used\_math; // 上一个使用过协处理器的进程。
143 extern struct task\_struct *current; // 当前进程结构指针变量。
144 extern long volatile jiffies; // 从开机开始算起的滴答数（10ms/滴答）。
145 extern long startup\_time; // 开机时间。从 1970:0:0:0 开始计时的秒数。
146
147 #define CURRENT\_TIME (startup\_time+jiffies/HZ) // 当前时间（秒数）。
148
149 // 添加定时器函数（定时时间 jiffies 滴答数，定时到时调用函数*fn()）。（ kernel/sched.c, 272）
150 extern void add\_timer(long jiffies, void (*fn)(void));
151 // 不可中断的等待睡眠。（ kernel/sched.c, 151 ）
152 extern void sleep\_on(struct task\_struct ** p);
153 // 可中断的等待睡眠。（ kernel/sched.c, 167 ）
154 extern void interruptible\_sleep\_on(struct task\_struct ** p);
155 // 明确唤醒睡眠的进程。（ kernel/sched.c, 188 ）
156 extern void wake\_up(struct task\_struct ** p);
157
158

```

```

149 /*
150  * Entry into gdt where to find first TSS. 0-nul, 1-cs, 2-ds, 3-syscall
151  * 4-TSS0, 5-LDT0, 6-TSS1 etc ...
152 */
/*
  * 寻找第 1 个 TSS 在全局表中的入口。0-没有用 nul, 1-代码段 cs, 2-数据段 ds, 3-系统段 syscall
  * 4-任务状态段 TSS0, 5-局部表 LTD0, 6-任务状态段 TSS1, 等。
  */
// 全局表中第 1 个任务状态段(TSS)描述符的选择符索引号。
153 #define FIRST_TSS_ENTRY 4
// 全局表中第 1 个局部描述符表(LDT)描述符的选择符索引号。
154 #define FIRST_LDT_ENTRY (FIRST_TSS_ENTRY+1)
// 宏定义, 计算在全局表中第 n 个任务的 TSS 描述符的索引号 (选择符)。
155 #define TSS(n) (((unsigned long) n)<<4)+(FIRST_TSS_ENTRY<<3))
// 宏定义, 计算在全局表中第 n 个任务的 LDT 描述符的索引号。
156 #define LDT(n) (((unsigned long) n)<<4)+(FIRST_LDT_ENTRY<<3))
// 宏定义, 加载第 n 个任务的寄存器 tr。
157 #define ltr(n) __asm__("ltr %%ax::\"a\" (TSS(n))")
// 宏定义, 加载第 n 个任务的局部描述符表寄存器 ldtr。
158 #define lldt(n) __asm__("lldt %%ax::\"a\" (LDT(n))")
// 取当前运行任务的编号 (是任务数组中的索引值, 与进程号 pid 不同)。
// 返回: n - 当前任务号。用于 (kernel/traps.c, 79)。
159 #define str(n) \
160 __asm__("str %%ax|n|t\" \           // 将任务寄存器中 TSS 段的有效地址→ax
161         \"subl %2, %%eax|n|t\" \     // (eax - FIRST_TSS_ENTRY*8)→eax
162         \"shrl $4, %%eax\" \         // (eax/16)→eax = 当前任务号。
163         : \"=a\" (n) \
164         : \"a\" (0), \"i\" (FIRST_TSS_ENTRY<<3))
165 /*
166  *      switch_to(n) should switch tasks to task nr n, first
167  * checking that n isn't the current task, in which case it does nothing.
168  * This also clears the TS-flag if the task we switched to has used
169  * the math co-processor latest.
170 */
/*
  * switch_to(n)将切换当前任务到任务 nr, 即 n。首先检测任务 n 不是当前任务,
  * 如果是则什么也不做退出。如果我们切换到任务最近 (上次运行) 使用过数学
  * 协处理器的话, 则还需复位控制寄存器 cr0 中的 TS 标志。
  */
// 输入: %0 - 新 TSS 的偏移地址(*&__tmp.a); %1 - 存放新 TSS 的选择符值(*&__tmp.b);
// dx - 新任务 n 的选择符; ecx - 新任务指针 task[n]。
// 其中临时数据结构 __tmp 中, a 的值是 32 位偏移值, b 为新 TSS 的选择符。在任务切换时, a 值
// 没有用 (忽略)。在判断新任务上次执行是否使用过协处理器时, 是通过将新任务状态段的地址与
// 保存在 last_task_used_math 变量中的使用过协处理器的任务状态段的地址进行比较而作出的。
171 #define switch_to(n) {\
172 struct {long a,b;} __tmp; \
173 __asm__("cmpl %%ecx, _current|n|t\" \ // 任务 n 是当前任务吗?(current ==task[n]?)
174         \"je 1f|n|t\" \             // 是, 则什么都不做, 退出。
175         \"movw %%dx, %1|n|t\" \     // 将新任务的选择符→*&__tmp.b。
176         \"xchgl %%ecx, _current|n|t\" \ // current = task[n]; ecx = 被切换出的任务。
177         \"ljmp %0|n|t\" \           // 执行长跳转至*&__tmp, 造成任务切换。
178         // 在任务切换回来后才会继续执行下面的语句。
179         \"cmpl %%ecx, _last_task_used_math|n|t\" \ // 新任务上次使用过协处理器吗?

```

```

179     "jne 1f|n|t" \                                // 没有则跳转, 退出。
180     "clts|n" \                                     // 新任务上次使用过协处理器, 则清 cr0 的 TS 标志。
181     "l:" \
182     :: "m" (*&__tmp.a), "m" (*&__tmp.b), \
183     "d" ( TSS(n)), "c" ((long) task[n])); \
184 }
185
// 页面地址对准。(在内核代码中没有任何地方引用!!)
186 #define PAGE_ALIGN(n) (((n)+0xfff)&0xfffff000)
187
// 设置位于地址 addr 处描述符中的各基地址字段(基地址是 base), 参见列表后说明。
// %0 - 地址 addr 偏移 2; %1 - 地址 addr 偏移 4; %2 - 地址 addr 偏移 7; edx - 基地址 base。
188 #define set_base(addr, base) \
189 __asm__( "movw %dx, %0|n|t" \                    // 基址 base 低 16 位(位 15-0) → [addr+2]。
190         "rorl $16, %%edx|n|t" \                  // edx 中基址高 16 位(位 31-16) → dx。
191         "movb %%dl, %1|n|t" \                    // 基址高 16 位中的低 8 位(位 23-16) → [addr+4]。
192         "movb %%dh, %2" \                        // 基址高 16 位中的高 8 位(位 31-24) → [addr+7]。
193         :: "m" (*(addr+2)), \
194         "m" (*(addr+4)), \
195         "m" (*(addr+7)), \
196         "d" (base) \
197         : "dx")
198
// 设置位于地址 addr 处描述符中的段限长字段(段长是 limit)。
// %0 - 地址 addr; %1 - 地址 addr 偏移 6 处; edx - 段长值 limit。
199 #define set_limit(addr, limit) \
200 __asm__( "movw %dx, %0|n|t" \                    // 段长 limit 低 16 位(位 15-0) → [addr]。
201         "rorl $16, %%edx|n|t" \                  // edx 中的段长高 4 位(位 19-16) → dl。
202         "movb %1, %%dh|n|t" \                    // 取原[addr+6]字节 → dh, 其中高 4 位是些标志。
203         "andb $0xf0, %%dh|n|t" \                 // 清 dh 的低 4 位(将存放段长的位 19-16)。
204         "orb %%dh, %%dl|n|t" \                   // 将原高 4 位标志和段长的高 4 位(位 19-16)合成 1 字节,
205         "movb %%dl, %1" \                         // 并放会[addr+6]处。
206         :: "m" (*(addr)), \
207         "m" (*(addr+6)), \
208         "d" (limit) \
209         : "dx")
210
// 设置局部描述符表中 ldt 描述符的基地址字段。
211 #define set_base(ldt, base) set_base((char *)&(ldt), base)
// 设置局部描述符表中 ldt 描述符的段长字段。
212 #define set_limit(ldt, limit) set_limit((char *)&(ldt), (limit-1)>>12)
213
// 从地址 addr 处描述符中取段基地址。功能与_set_base()正好相反。
// edx - 存放基地址(__base); %1 - 地址 addr 偏移 2; %2 - 地址 addr 偏移 4; %3 - addr 偏移 7。
214 #define get_base(addr) ({
215 unsigned long __base; \
216 __asm__( "movb %3, %%dh|n|t" \                    // 取[addr+7]处基址高 16 位的高 8 位(位 31-24) → dh。
217         "movb %2, %%dl|n|t" \                    // 取[addr+4]处基址高 16 位的低 8 位(位 23-16) → dl。
218         "shll $16, %%edx|n|t" \                  // 基址高 16 位移到 edx 中高 16 位处。
219         "movw %1, %dx" \                          // 取[addr+2]处基址低 16 位(位 15-0) → dx。
220         : "=d" (__base) \
221         : "m" (*(addr+2)), \
222         "m" (*(addr+4)), \

```

```

223         "m" (*((addr)+7)); \
224 __base;})
225
226 // 取局部描述符表中 ldt 所指段描述符中的基地址。
227 #define get_base(ldt)  _get_base( ((char *)&(ldt)) )
228
229 // 取段选择符 segment 的段长值。
230 // %0 - 存放段长值(字节数); %1 - 段选择符 segment。
231 #define get_limit(segment) ({ \
232 unsigned long __limit; \
233 __asm__( "lsl %1,%0\n\tincl %0": "=r" (__limit): "r" (segment)); \
234 __limit;})
235 #endif
236

```

11.29.3 其它信息

11.29.3.1 任务状态段信息

31	23	15	7	0	
I/O 映射图基地址 (MAP BASE)					64
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0					局部描述符表 (LDT) 的选择符
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0					GS
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0					FS
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0					DS
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0					SS
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0					CS
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0					ES
EDI					44
ESI					40
EBP					3C
ESP					38
EBX					34
EDX					30
ECX					2C
EAX					28
EFLAGS					24
指令指针 (EIP)					20
页目录基地址寄存器 CR3 (PDBR)					1C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0					SS2
ESP2					14
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0					SS1
ESP1					0C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0					SS0
ESP0					04
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0					前一执行任务 TSS 的描述符
					00

任务状态段的详细说明请参考附录。这里对其进行简单描述。

CPU 管理任务需要的所有信息被存储于一个特殊类型的段中，任务状态段(task state segment - TSS)。图中显示出执行 80386 任务的 TSS 格式。

TSS 中的字段可以分为两类：

1. CPU 在进行任务切换时更新的动态信息集。这些字段有：
 - o 通用寄存器 (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI);
 - o 段寄存器 (ES, CS, SS, DS, FS, GS);
 - o 标志寄存器 (EIP);
 - o 指令指针 (EIP);
 前一个执行任务的 TSS 的选择符 (仅当返回时才更新)。
2. CPU 读取但不会更改的静态信息集。这些字段有：
 - o 任务的 LDT 的选择符;
 - o 含有任务页目录基地址的寄存器 (PDBR);
 - o 特权级 0-2 的堆栈指针;
 - o 当任务进行切换时导致 CPU 产生一个调试(debug)异常的 T-比特位 (调试跟踪位);
 - o I/O 比特位图基地址 (其长度上限就是 TSS 的长度上限, 在 TSS 描述符中说明)。

任务状态段可以存放在线性空间的任何地方。与其它各类段相似，任务状态段也是由描述符来定义的。当前正在执行任务的 TSS 是由任务寄存器 (TR) 来指示的。指令 LTR 和 STR 用来修改和读取任务寄存器中的选择符 (任务寄存器的可见部分)。

I/O 比特位图中的每 1 比特对应 1 个 I/O 端口。比如端口 41 的比特位就是 I/O 位图基地址+5, 位偏移 1 处。在保护模式中，当遇到 1 个 I/O 指令时(IN, INS, OUT, OUTS), CPU 首先就会检查当前特权级是否小于标志寄存器的 IOPL, 如果这个条件满足，就执行该 I/O 操作。如果不满足，那么 CPU 就会检查 TSS 中的 I/O 比特位图。如果相应比特位是置位的，就会产生一般保护性异常，否则就会执行该 I/O 操作。

11.29.3.2 段描述符

参见附录。

11.30 sys.h 文件

11.30.1 功能描述

sys.h 头文件列出了内核中所有系统调用函数的原型，以及系统调用函数指针表。

11.30.2 代码注释

列表 linux/include/linux/sys.h 文件

1	extern int sys_setup ();	// 系统启动初始化设置函数。	(kernel/blk_drv/hd.c, 71)
2	extern int sys_exit ();	// 程序退出。	(kernel/exit.c, 137)

3	extern int sys fork() ;	// 创建进程。	(kernel/system_call.s, 208)
4	extern int sys read() ;	// 读文件。	(fs/read_write.c, 55)
5	extern int sys write() ;	// 写文件。	(fs/read_write.c, 83)
6	extern int sys open() ;	// 打开文件。	(fs/open.c, 138)
7	extern int sys close() ;	// 关闭文件。	(fs/open.c, 192)
8	extern int sys waitpid() ;	// 等待进程终止。	(kernel/exit.c, 142)
9	extern int sys creat() ;	// 创建文件。	(fs/open.c, 187)
10	extern int sys link() ;	// 创建一个文件的硬连接。	(fs/namei.c, 721)
11	extern int sys unlink() ;	// 删除一个文件名(或删除文件)。	(fs/namei.c, 663)
12	extern int sys execve() ;	// 执行程序。	(kernel/system_call.s, 200)
13	extern int sys chdir() ;	// 更改当前目录。	(fs/open.c, 75)
14	extern int sys time() ;	// 取当前时间。	(kernel/sys.c, 102)
15	extern int sys mknod() ;	// 建立块/字符特殊文件。	(fs/namei.c, 412)
16	extern int sys chmod() ;	// 修改文件属性。	(fs/open.c, 105)
17	extern int sys chown() ;	// 修改文件宿主和所属组。	(fs/open.c, 121)
18	extern int sys break() ;	//	(-kernel/sys.c, 21)
19	extern int sys stat() ;	// 使用路径名取文件的状态信息。	(fs/stat.c, 36)
20	extern int sys lseek() ;	// 重新定位读/写文件偏移。	(fs/read_write.c, 25)
21	extern int sys getpid() ;	// 取进程 id。	(kernel/sched.c, 348)
22	extern int sys mount() ;	// 安装文件系统。	(fs/super.c, 200)
23	extern int sys umount() ;	// 卸载文件系统。	(fs/super.c, 167)
24	extern int sys setuid() ;	// 设置进程用户 id。	(kernel/sys.c, 143)
25	extern int sys getuid() ;	// 取进程用户 id。	(kernel/sched.c, 358)
26	extern int sys stime() ;	// 设置系统时间日期。	(-kernel/sys.c, 148)
27	extern int sys ptrace() ;	// 程序调试。	(-kernel/sys.c, 26)
28	extern int sys alarm() ;	// 设置报警。	(kernel/sched.c, 338)
29	extern int sys fstat() ;	// 使用文件句柄取文件的状态信息。	(fs/stat.c, 47)
30	extern int sys pause() ;	// 暂停进程运行。	(kernel/sched.c, 144)
31	extern int sys utime() ;	// 改变文件的访问和修改时间。	(fs/open.c, 24)
32	extern int sys stty() ;	// 修改终端行设置。	(-kernel/sys.c, 31)
33	extern int sys gtty() ;	// 取终端行设置信息。	(-kernel/sys.c, 36)
34	extern int sys access() ;	// 检查用户对一个文件的访问权限。	(fs/open.c, 47)
35	extern int sys nice() ;	// 设置进程执行优先权。	(kernel/sched.c, 378)
36	extern int sys ftime() ;	// 取日期和时间。	(-kernel/sys.c, 16)
37	extern int sys sync() ;	// 同步高速缓冲与设备中数据。	(fs/buffer.c, 44)
38	extern int sys kill() ;	// 终止一个进程。	(kernel/exit.c, 60)
39	extern int sys rename() ;	// 更改文件名。	(-kernel/sys.c, 41)
40	extern int sys mkdir() ;	// 创建目录。	(fs/namei.c, 463)
41	extern int sys rmdir() ;	// 删除目录。	(fs/namei.c, 587)
42	extern int sys dup() ;	// 复制文件句柄。	(fs/fcntl.c, 42)
43	extern int sys pipe() ;	// 创建管道。	(fs/pipe.c, 71)
44	extern int sys times() ;	// 取运行时间。	(kernel/sys.c, 156)
45	extern int sys prof() ;	// 程序执行时间区域。	(-kernel/sys.c, 46)
46	extern int sys brk() ;	// 修改数据段长度。	(kernel/sys.c, 168)
47	extern int sys setgid() ;	// 设置进程组 id。	(kernel/sys.c, 72)
48	extern int sys getgid() ;	// 取进程组 id。	(kernel/sched.c, 368)
49	extern int sys signal() ;	// 信号处理。	(kernel/signal.c, 48)
50	extern int sys geteuid() ;	// 取进程有效用户 id。	(kernel/sched.c, 363)
51	extern int sys getegid() ;	// 取进程有效组 id。	(kernel/sched.c, 373)
52	extern int sys acct() ;	// 进程记帐。	(-kernel/sys.c, 77)
53	extern int sys phys() ;	//	(-kernel/sys.c, 82)
54	extern int sys lock() ;	//	(-kernel/sys.c, 87)
55	extern int sys ioctl() ;	// 设备控制。	(fs/ioctl.c, 30)

```

56 extern int sys\_fcntl();           // 文件句柄操作。           (fs/fcntl.c, 47)
57 extern int sys\_mpx();             //                        (-kernel/sys.c, 92)
58 extern int sys\_setpgid();          // 设置进程组 id。           (kernel/sys.c, 181)
59 extern int sys\_ulimit();           //                        (-kernel/sys.c, 97)
60 extern int sys\_uname();            // 显示系统信息。           (kernel/sys.c, 216)
61 extern int sys\_umask();            // 取默认文件创建属性码。   (kernel/sys.c, 230)
62 extern int sys\_chroot();           // 改变根系统。             (fs/open.c, 90)
63 extern int sys\_ustat();            // 取文件系统信息。         (fs/open.c, 19)
64 extern int sys\_dup2();             // 复制文件句柄。           (fs/fcntl.c, 36)
65 extern int sys\_getppid();          // 取父进程 id。            (kernel/sched.c, 353)
66 extern int sys\_getpgrp();          // 取进程组 id, 等于 getpgid(0)。 (kernel/sys.c, 201)
67 extern int sys\_setsid();          // 在新会话中运行程序。     (kernel/sys.c, 206)
68 extern int sys\_sigaction();       // 改变信号处理过程。       (kernel/signal.c, 63)
69 extern int sys\_sgetmask();         // 取信号屏蔽码。           (kernel/signal.c, 15)
70 extern int sys\_ssetmask();         // 设置信号屏蔽码。         (kernel/signal.c, 20)
71 extern int sys\_setreuid();         // 设置真实与/或有效用户 id。 (kernel/sys.c, 118)
72 extern int sys\_setregid();         // 设置真实与/或有效组 id。  (kernel/sys.c, 51)
73
// 系统调用函数指针表。用于系统调用中断处理程序(int 0x80), 作为跳转表。
74 fn_ptr sys\_call\_table[] = { sys\_setup, sys\_exit, sys\_fork, sys\_read,
75 sys\_write, sys\_open, sys\_close, sys\_waitpid, sys\_creat, sys\_link,
76 sys\_unlink, sys\_execve, sys\_chdir, sys\_time, sys\_mknod, sys\_chmod,
77 sys\_chown, sys\_break, sys\_stat, sys\_lseek, sys\_getpid, sys\_mount,
78 sys\_umount, sys\_setuid, sys\_getuid, sys\_stime, sys\_ptrace, sys\_alarm,
79 sys\_fstat, sys\_pause, sys\_utime, sys\_stty, sys\_gtty, sys\_access,
80 sys\_nice, sys\_ftime, sys\_sync, sys\_kill, sys\_rename, sys\_mkdir,
81 sys\_rmdir, sys\_dup, sys\_pipe, sys\_times, sys\_prof, sys\_brk, sys\_setgid,
82 sys\_getgid, sys\_signal, sys\_geteuid, sys\_getegid, sys\_acct, sys\_phys,
83 sys\_lock, sys\_ioctl, sys\_fcntl, sys\_mpx, sys\_setpgid, sys\_ulimit,
84 sys\_uname, sys\_umask, sys\_chroot, sys\_ustat, sys\_dup2, sys\_getppid,
85 sys\_getpgrp, sys\_setsid, sys\_sigaction, sys\_sgetmask, sys\_ssetmask,
86 sys\_setreuid, sys\_setregid };
87

```

11.31 tty.h 文件

11.31.1 功能描述

11.31.2 代码注释

列表 linux/include/linux/tty.h 文件

```

1 /*
2  * 'tty.h' defines some structures used by tty_io.c and some defines.
3  *
4  * NOTE! Don't touch this without checking that nothing in rs_io.s or
5  * con_io.s breaks. Some constants are hardwired into the system (mainly
6  * offsets into 'tty_queue'
7  */

```

```

8  /*
   * 'tty.h' 中定义了 tty_io.c 程序使用的某些结构和其它一些定义。
   *
   * 注意！在修改这里的定义时，一定要检查 rs_io.s 或 con_io.s 程序中不会出现问题。
   * 在系统中有些常量是直接写在程序中的（主要是一些 tty_queue 中的偏移值）。
   */
9  #ifndef TTY_H
10 #define TTY_H
11
12 #include <termios.h>          // 终端输入输出函数头文件。主要定义控制异步通信口的终端接口。
13
14 #define TTY_BUF_SIZE 1024    // tty 缓冲区大小。
15
16 // tty 等待队列数据结构。
17 struct tty_queue {
18     unsigned long data;        // 等待队列缓冲区中当前数据指针字符数[??]）。
19                                // 对于串口终端，则存放串行端口地址。
20     unsigned long head;       // 缓冲区中数据头指针。
21     unsigned long tail;       // 缓冲区中数据尾指针。
22     struct task_struct * proc_list; // 等待进程列表。
23     char buf[TTY_BUF_SIZE];    // 队列的缓冲区。
24 };
25
26 // 以下定义了 tty 等待队列中缓冲区操作宏函数。（tail 在前，head 在后）。
27 // a 缓冲区指针前移 1 字节，并循环。
28 #define INC(a) ((a) = ((a)+1) & (TTY_BUF_SIZE-1))
29 // a 缓冲区指针后退 1 字节，并循环。
30 #define DEC(a) ((a) = ((a)-1) & (TTY_BUF_SIZE-1))
31 // 清空指定队列的缓冲区。
32 #define EMPTY(a) ((a).head == (a).tail)
33 // 缓冲区还可存放字符的长度（空闲区长度）。
34 #define LEFT(a) (((a).tail-(a).head-1)&(TTY_BUF_SIZE-1))
35 // 缓冲区中最后一个位置。
36 #define LAST(a) ((a).buf[(TTY_BUF_SIZE-1)&((a).head-1)])
37 // 缓冲区满（如果为 1 的话）。
38 #define FULL(a) (!LEFT(a))
39 // 缓冲区中已存放字符的长度。
40 #define CHARS(a) (((a).head-(a).tail)&(TTY_BUF_SIZE-1))
41 // 从 queue 队列项缓冲区中取一字符（从 tail 处，并且 tail+=1）。
42 #define GETCH(queue, c) \
43 (void) ({c=(queue).buf[(queue).tail];INC((queue).tail);})
44 // 往 queue 队列项缓冲区中放置一字符（在 head 处，并且 head+=1）。
45 #define PUTCH(c, queue) \
46 (void) ({(queue).buf[(queue).head]=(c);INC((queue).head);})
47
48 // 判断终端键盘字符类型。
49 #define INTR_CHAR(tty) ((tty)->termios.c_cc[VINTR])    // 中断符。
50 #define QUIT_CHAR(tty) ((tty)->termios.c_cc[VQUIT])    // 退出符。
51 #define ERASE_CHAR(tty) ((tty)->termios.c_cc[VERASE])  // 删除符。
52 #define KILL_CHAR(tty) ((tty)->termios.c_cc[VKILL])    // 终止符。
53 #define EOF_CHAR(tty) ((tty)->termios.c_cc[VEOF])     // 文件结束符。
54 #define START_CHAR(tty) ((tty)->termios.c_cc[VSTART])  // 开始符。

```






```

42 #define STOP_CHAR(tty) ((tty)->termios.c_cc[VSTOP]) // 结束符。
43 #define SUSPEND_CHAR(tty) ((tty)->termios.c_cc[VSUSP]) // 挂起符。
44
// tty 数据结构。
45 struct tty_struct {
46     struct termios termios; // 终端 io 属性和控制字符数据结构。
47     int pgrp; // 所属进程组。
48     int stopped; // 停止标志。
49     void (*write)(struct tty_struct * tty); // tty 写函数指针。
50     struct tty_queue read_q; // tty 读队列。
51     struct tty_queue write_q; // tty 写队列。
52     struct tty_queue secondary; // tty 辅助队列(存放规范模式字符序列),
53     }; // 可称为规范(熟)模式队列。
54
55 extern struct tty_struct tty_table[]; // tty 结构数组。
56
57 /*      intr=^C      quit=^/      erase=del      kill=^U
58      eof=^D      vtime=\0      vmin=\1      sxtc=\0
59      start=^Q      stop=^S      susp=^Z      eol=\0
60      reprint=^R      discard=^U      werase=^W      lnext=^V
61      eol2=\0
62 */
/* 中断 intr=^C      退出 quit=^|      删除 erase=del      终止 kill=^U
* 文件结束 eof=^D      vtime=\0      vmin=\1      sxtc=\0
* 开始 start=^Q      停止 stop=^S      挂起 susp=^Z      行结束 eol=\0
* 重显 reprint=^R      丢弃 discard=^U      werase=^W      lnext=^V
* 行结束 eol2=\0
*/
// 控制字符对应的 ASCII 码值。[8 进制]
63 #define INIT_C_CC "\003\034\177\025\004\0\1\0\021\023\032\0\022\017\027\026\0"
64
65 void rs_init(void); // 异步串行通信初始化。(kernel/chr_drv/serial.c, 37)
66 void con_init(void); // 控制终端初始化。(kernel/chr_drv/console.c, 617)
67 void tty_init(void); // tty 初始化。(kernel/chr_drv/tty_io.c, 105)
68
69 int tty_read(unsigned c, char * buf, int n); // (kernel/chr_drv/tty_io.c, 230)
70 int tty_write(unsigned c, char * buf, int n); // (kernel/chr_drv/tty_io.c, 290)
71
72 void rs_write(struct tty_struct * tty); // (kernel/chr_drv/serial.c, 53)
73 void con_write(struct tty_struct * tty); // (kernel/chr_drv/console.c, 445)
74
75 void copy_to_cooked(struct tty_struct * tty); // (kernel/chr_drv/tty_io.c, 145)
76
77 #endif
78

```

11.32 include/sys/目录中的文件

列表 linux/include/sys/目录下的文件

Name	Size	Last modified (GMT)	Description
 stat.h	1304 bytes	1991-09-17 15:02:48	m
 times.h	200 bytes	1991-09-17 15:03:06	m
 types.h	805 bytes	1991-09-17 15:02:55	m
 utsname.h	234 bytes	1991-09-17 15:03:23	m
 wait.h	560 bytes	1991-09-17 15:06:07	m

11.33 stat.h 文件

11.33.1 功能描述

该头文件说明了函数 `stat()` 返回的数据及其结构类型，以及一些属性操作测试宏、函数原型。

11.33.2 代码注释

列表 linux/include/sys/stat.h 文件

```

1 #ifndef SYS\_STAT\_H
2 #define SYS\_STAT\_H
3
4 #include <sys/types.h>
5
6 struct stat {
7     dev\_t    st_dev;    // 含有文件的设备号。
8     ino\_t    st_ino;    // 文件 i 节点号。
9     umode\_t  st_mode;   // 文件属性（见下面）。
10    nlink\_t  st_nlink;  // 指定文件的连接数。
11    uid\_t    st_uid;    // 文件的用户（标识）号。
12    gid\_t    st_gid;    // 文件的组号。
13    dev\_t    st_rdev;   // 设备号（如果文件是特殊的字符文件或块文件）。
14    off\_t    st_size;   // 文件大小（字节数）（如果文件是常规文件）。
15    time\_t   st_atime;  // 上次（最后）访问时间。

```

```

16     time_t  st_mtime;    // 最后修改时间。
17     time_t  st_ctime;    // 最后节点修改时间。
18 };
19
    // 以下这些是 st_mode 值的符号名称。
    // 文件类型：
20 #define S_IFMT 00170000    // 文件类型。
21 #define S_IFREG 0100000    // 常规文件。
22 #define S_IFBLK 0060000    // 块特殊（设备）文件。
23 #define S_IFDIR 0040000    // 目录文件。
24 #define S_IFCHR 0020000    // 字符设备文件。
25 #define S_IFIFO 0010000    // FIFO 特殊文件。
    // 文件属性位：
26 #define S_ISUID 0004000    // 执行时设置用户 ID (set-user-ID)。
27 #define S_ISGID 0002000    // 执行时设置组 ID。
28 #define S_ISVTX 0001000    // 对于目录，受限删除标志。
29
30 #define S_ISREG(m)        (((m) & S_IFMT) == S_IFREG)    // 测试是否常规文件。
31 #define S_ISDIR(m)        (((m) & S_IFMT) == S_IFDIR)    // 是否目录文件。
32 #define S_ISCHR(m)        (((m) & S_IFMT) == S_IFCHR)    // 是否字符设备文件。
33 #define S_ISBLK(m)        (((m) & S_IFMT) == S_IFBLK)    // 是否块设备文件。
34 #define S_ISFIFO(m)       (((m) & S_IFMT) == S_IFIFO)    // 是否 FIFO 特殊文件。
35
36 #define S_IRWXU 00700      // 宿主可以读、写、执行/搜索。
37 #define S_IRUSR 00400      // 宿主读许可。
38 #define S_IWUSR 00200      // 宿主写许可。
39 #define S_IXUSR 00100      // 宿主执行/搜索许可。
40
41 #define S_IRWXG 00070      // 组成员可以读、写、执行/搜索。
42 #define S_IRGRP 00040      // 组成员读许可。
43 #define S_IWGRP 00020      // 组成员写许可。
44 #define S_IXGRP 00010      // 组成员执行/搜索许可。
45
46 #define S_IRWXO 00007      // 其他人读、写、执行/搜索许可。
47 #define S_IROTH 00004      // 其他人读许可。
48 #define S_IWOTH 00002      // 其他人写许可。
49 #define S_IXOTH 00001      // 其他人执行/搜索许可。
50
51 extern int chmod(const char *_path, mode_t mode);    // 修改文件属性。
52 extern int fstat(int fildes, struct stat *stat_buf); // 取指定文件句柄的文件状态信息。
53 extern int mkdir(const char *_path, mode_t mode);    // 创建目录。
54 extern int mkfifo(const char *_path, mode_t mode);    // 创建管道文件。
55 extern int stat(const char *filename, struct stat *stat_buf); // 取指定文件名的文件状态信息。
56 extern mode_t umask(mode_t mask);    // 设置属性屏蔽码。
57
58 #endif
59

```

11.34 times.h 文件

11.34.1 功能描述

该头文件中主要定义了文件访问与修改时间结构 `tms`。它将由 `times()` 函数返回。其中 `time_t` 是在 `sys/types.h` 中定义的。还定义了一个函数原型 `times()`。

11.34.2 代码注释

列表 linux/include/sys/times.h 文件

```

1 #ifndef TIMES\_H
2 #define TIMES\_H
3
4 #include <sys/types.h>    // 类型头文件。定义了基本的系统数据类型。
5
6 struct tms {
7     time\_t tms_ftime; // 用户使用的 CPU 时间。
8     time\_t tms_stime; // 系统（内核）CPU 时间。
9     time\_t tms_cutime; // 已终止的子进程使用的用户 CPU 时间。
10    time\_t tms_cstime; // 已终止的子进程使用的系统 CPU 时间。
11 };
12
13 extern time\_t times(struct tms * tp);
14
15 #endif
16

```

11.35 types.h 文件

11.35.1 功能描述

`types.h` 头文件中定义了基本的数据类型。所有的类型定义为适当的数学类型长度。另外，`size_t` 是无符号整数类型，`off_t` 是扩展的符号整数类型，`pid_t` 是符号整数类型。

11.35.2 代码注释

列表 linux/include/sys/types.h 文件

```

1 #ifndef SYS\_TYPES\_H
2 #define SYS\_TYPES\_H
3
4 #ifndef SIZE\_T
5 #define SIZE\_T
6 typedef unsigned int size\_t;    // 用于对象的大小（长度）。
7 #endif
8
9 #ifndef TIME\_T

```

```

10 #define \_TIME\_T
11 typedef long time\_t; // 用于时间（以秒计）。
12 #endif
13
14 #ifndef PTRDIFF\_T
15 #define PTRDIFF\_T
16 typedef long ptrdiff\_t;
17 #endif
18
19 #ifndef NULL
20 #define NULL ((void *) 0)
21 #endif
22
23 typedef int pid\_t; // 用于进程号和进程组号。
24 typedef unsigned short uid\_t; // 用于用户号（用户标识号）。
25 typedef unsigned char gid\_t; // 用于组号。
26 typedef unsigned short dev\_t; // 用于设备号。
27 typedef unsigned short ino\_t; // 用于文件序列号。
28 typedef unsigned short mode\_t; // 用于某些文件属性。
29 typedef unsigned short umode\_t; //
30 typedef unsigned char nlink\_t; // 用于连接计数。
31 typedef int daddr\_t;
32 typedef long off\_t; // 用于文件长度（大小）。
33 typedef unsigned char u\_char; // 无符号字符类型。
34 typedef unsigned short ushort; // 无符号短整数类型。
35
36 typedef struct { int quot,rem; } div\_t; // 用于 DIV 操作。
37 typedef struct { long quot,rem; } ldiv\_t; // 用于长 DIV 操作。
38
39 struct ustat {
40     daddr\_t f_tfree;
41     ino\_t f_tinode;
42     char f_fname[6];
43     char f_fpack[6];
44 };
45
46 #endif
47

```

11.36 utsname.h 文件

11.36.1 功能描述

utsname.h 是系统名称结构头文件。其中定义了结构 utsname 以及函数原型 uname()。POSIX 要求字符串数组长度应该是不指定的，但是其中存储的数据需以 null 终止。因此该版内核的 utsname 结构定义不符合要求（数组长度都被定义为 9）。

11.36.2 代码注释

列表 linux/include/sys/utsname.h 文件

```

1 #ifndef \_SYS\_UTSNAME\_H
2 #define \_SYS\_UTSNAME\_H
3
4 #include <sys/types.h>    // 类型头文件。定义了基本的系统数据类型。
5
6 struct utsname {
7     char sysname[9];    // 本版本操作系统的名称。
8     char nodename[9];    // 与实现相关的网络中节点名称。
9     char release[9];    // 本实现的当前发行级别。
10    char version[9];    // 本次发行的版本级别。
11    char machine[9];    // 系统运行的硬件类型名称。
12 };
13
14 extern int uname(struct utsname * utsbuf);
15
16 #endif
17

```

11.37 wait.h 文件

11.37.1 功能描述

该头文件描述了进程等待时信息。包括一些符号常数和 wait()、waitpid() 函数原型申明。

11.37.2 代码注释

列表 linux/include/sys/wait.h 文件

```

1 #ifndef \_SYS\_WAIT\_H
2 #define \_SYS\_WAIT\_H
3
4 #include <sys/types.h>
5
6 #define LOW(v)          ( (v) & 0377)          // 取低字节（8 进制表示）。
7 #define HIGH(v)        ( ((v) >> 8) & 0377)    // 取高字节。
8
9 /* options for waitpid, WUNTRACED not supported */
10 /* waitpid 的选项，其中 WUNTRACED 未被支持 */
11 #define WNOHANG         1          // 如果没有状态也不要挂起，并立刻返回。
12 #define WUNTRACED       2          // 报告停止执行的子进程状态。
13
14 #define WIFEXITED(s)     (!((s)&0xFF)          // 如果子进程正常退出，则为真。
15 #define WIFSTOPPED(s)   (((s)&0xFF)==0x7F)    // 如果子进程正停止着，则为 true。
16 #define WEXITSTATUS(s)  (((s)>>8)&0xFF)        // 返回退出状态。
17 #define WTERMSIG(s)      ((s)&0x7F)           // 返回导致进程终止的信号值（信号量）。
18 #define WSTOPSIG(s)      (((s)>>8)&0xFF)        // 返回导致进程停止的信号值。
19 #define WIFSIGNALED(s)   (((unsigned int)(s)-1 & 0xFFFF) < 0xFF) // 如果由于未捕捉到信号
                                                                    // 而导致子进程退出则为真。
20
21 // wait() 和 waitpid() 函数允许进程获取与其子进程之一的状态信息。各种选项允许获取已经终止或

```

```
// 停止的子进程状态信息。如果存在两个或两个以上子进程的状态信息，则报告的顺序是不指定的。  
// wait() 将挂起当前进程，直到其子进程之一退出（终止），或者收到要求终止该进程的信号，  
// 或者是需要调用一个信号句柄（信号处理程序）。  
// waitpid() 挂起当前进程，直到 pid 指定的子进程退出（终止）或者收到要求终止该进程的信号，  
// 或者是需要调用一个信号句柄（信号处理程序）。  
// 如果 pid=-1, options=0, 则 waitpid() 的作用与 wait() 函数一样。否则其行为将随 pid 和 options  
// 参数的不同而不同。（参见 kernel/exit.c, 142）  
20 pid_t wait(int *stat_loc);  
21 pid_t waitpid(pid_t pid, int *stat_loc, int options);  
22  
23 #endif  
24
```
