```
/*
 *  linux/kernel/sched.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * 'sched.c' is the main kernel file. It contains scheduling primitives
 * (sleep_on, wakeup, schedule etc) as well as a number of simple system
 * call functions (type getpid(), which just extracts a field from
 * current-task
 */
/*
 * 'sched.c'是主要的内核文件。其中包括有关调度的基本函数(sleep_on、wakeup、schedule 等)
 * 以及一些简单的系统调用函数（比如 getpid()，仅从当前任务中获取一个字段）。
 */

// 下面是调度程序头文件。定义了任务结构 task_struct、第 1 个初始任务的数据。还有一些以宏
// 的形式定义的有关描述符参数设置和获取的嵌入式汇编函数程序。
#include <linux/sched.h>
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
#include <linux/sys.h>    // 系统调用头文件。含有 82 个系统调用 C 函数程序,以'sys_'开头。
#include <linux/fdreg.h>  // 软驱头文件。含有软盘控制器参数的一些定义。
#include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
#include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。
#include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。

#include <signal.h>       // 信号头文件。定义信号符号常量，sigaction 结构，操作函数原型。

// 该宏取信号 nr 在信号位图中对应位的二进制数值。信号编号 1-32。比如信号 5 的位图数值等于
// 1<<(5-1) = 16 = 00010000b。
#define _S(nr) (1<<((nr)-1))
// 除了 SIGKILL 和 SIGSTOP 信号以外其他信号都是可阻塞的(…1011,1111,1110,1111,1111b)。
#define _BLOCKABLE (~(_S(SIGKILL) | _S(SIGSTOP)))

// 内核调试函数。显示任务号 nr 的进程号、进程状态和内核堆栈空闲字节数（大约）。
void show_task(int nr,struct task_struct * p)
{
        int i,j = 4096-sizeof(struct task_struct);

        printk("%d: pid=%d, state=%d, father=%d, child=%d, ",nr,p->pid,
                p->state, p->p_pptr->pid, p->p_cptr ? p->p_cptr->pid : -1);
        i=0;
        while (i<j && !((char *)(p+1))[i])  // 检测指定任务数据结构以后等于 0 的字节数。
                i++;
        printk("%d/%d chars free in kstack\n\r",i,j);
        printk("   PC=%08X.", *(1019 + (unsigned long *) p));
        if (p->p_ysptr || p->p_osptr)
                printk("   Younger sib=%d, older sib=%d\n\r",
                        p->p_ysptr ? p->p_ysptr->pid : -1,
                        p->p_osptr ? p->p_osptr->pid : -1);
        else
```

```
42                 printk("\n\r");
43 }
44
   // 显示所有任务的任务号、进程号、进程状态和内核堆栈空闲字节数（大约）。
   // NR_TASKS 是系统能容纳的最大进程(任务)数量(64 个)，定义在 include/kernel/sched.h 第 6 行。
45 void show_state(void)
46 {
47         int i;
48
49         printk("\rTask-info:\n\r");
50         for (i=0;i<NR_TASKS;i++)
51                 if (task[i])
52                         show_task(i,task[i]);
53 }
54
   // PC 机 8253 定时芯片的输入时钟频率约为 1.193180MHz。Linux 内核希望定时器发出中断的频率是
   // 100Hz，也即每 10ms 发出一次时钟中断。因此这里 LATCH 是设置 8253 芯片的初值，参见 438 行。
55 #define LATCH (1193180/HZ)
56
57 extern void mem_use(void);          // [??]没有任何地方定义和引用该函数。
58
59 extern int timer_interrupt(void);  // 时钟中断处理程序（kernel/system_call.s，176）。
60 extern int system_call(void);      // 系统调用中断处理程序（kernel/system_call.s，80）。
61
   // 每个任务（进程）在内核态运行时都有自己的内核态堆栈。这里定义了任务的内核态堆栈结构。
   // 这里定义任务联合（任务结构成员和 stack 字符数组成员）。因为一个任务的数据结构与其内核
   // 态堆栈放在同一内存页中，所以从堆栈段寄存器 ss 可以获得其数据段选择符。
62 union task_union {
63         struct task_struct task;
64         char stack[PAGE_SIZE];
65 };
66
   // 设置初始任务的数据。初始数据在 include/kernel/sched.h 中，第 156 行开始。
67 static union task_union init_task = {INIT_TASK,};
68
   // 从开机开始算起的滴答数时间值全局变量（10ms/滴答）。系统时钟中断每发生一次即一个滴答。
   // 前面的限定符 volatile，英文解释是易改变的、不稳定的意思。这个限定词的含义是向编译器
   // 指明变量的内容可能会由于被其他程序修改而变化。通常在程序中申明一个变量时， 编译器会
   // 尽量把它存放在通用寄存器中，例如 ebx，以提高访问效率。当 CPU 把其值放到 ebx 中后一般
   // 就不会再关心该变量对应内存位置中的内容。若此时其他程序（例如内核程序或一个中断过程）
   // 修改了内存中该变量的值，ebx 中的值并不会随之更新。为了解决这种情况就创建了 volatile
   // 限定符，让代码在引用该变量时一定要从指定内存位置中取得其值。这里即是要求 gcc 不要对
   // jiffies 进行优化处理，也不要挪动位置，并且需要从内存中取其值。因为时钟中断处理过程
   // 等程序会修改它的值。
69 unsigned long volatile jiffies=0;
70 unsigned long startup_time=0;                  // 开机时间。从 1970:0:0:0 开始计时的秒数。
   // 这个变量用于累计需要调整地时间嘀嗒数。
71 int jiffies_offset = 0;          /* # clock ticks to add to get "true
72                                     time".  Should always be less than
73                                     1 second's worth.  For time fanatics
74                                     who like to syncronize their machines
75                                     to WWV :-) */
   /* 为调整时钟而需要增加的时钟嘀嗒数，以获得“精确时间”。这些调整用嘀嗒数
```

```
 * 的总和不应该超过 1 秒。这样做是为了那些对时间精确度要求苛刻的人，他们喜
 * 欢自己的机器时间与 WWV 同步 :-)
 */

struct task_struct *current = &(init_task.task);  // 当前任务指针（初始化指向任务 0）。
struct task_struct *last_task_used_math = NULL;   // 使用过协处理器任务的指针。

// 定义任务指针数组。第 1 项被初始化指向初始任务（任务 0）的任务数据结构。
struct task_struct * task[NR_TASKS] = {&(init_task.task), };

// 定义用户堆栈，共 1K 项，容量 4K 字节。在内核初始化操作过程中被用作内核栈，初始化完成
// 以后将被用作任务 0 的用户态堆栈。在运行任务 0 之前它是内核栈，以后用作任务 0 和 1 的用
// 户态栈。下面结构用于设置堆栈 ss:esp（数据段选择符，指针），见 head.s，第 23 行。
// ss 被设置为内核数据段选择符（0x10），指针 esp 指在 user_stack 数组最后一项后面。这是
// 因为 Intel CPU 执行堆栈操作时是先递减堆栈指针 sp 值，然后在 sp 指针处保存入栈内容。
long user_stack [ PAGE_SIZE>>2 ] ;

struct {
        long * a;
        short b;
        } stack_start = { & user_stack [PAGE_SIZE>>2] , 0x10 };
/*
 *  'math_state_restore()' saves the current math information in the
 * old math state array, and gets the new ones from the current task
 */
/*
 * 将当前协处理器内容保存到老协处理器状态数组中，并将当前任务的协处理器
 * 内容加载进协处理器。
 */
// 当任务被调度交换过以后，该函数用以保存原任务的协处理器状态（上下文）并恢复新调度进
// 来的当前任务的协处理器执行状态。
void math_state_restore()
{
// 如果任务没变则返回(上一个任务就是当前任务)。这里"上一个任务"是指刚被交换出去的任务。
        if (last_task_used_math == current)
                return;
// 在发送协处理器命令之前要先发 WAIT 指令。如果上个任务使用了协处理器，则保存其状态。
        __asm__("fwait");
        if (last_task_used_math) {
                __asm__("fnsave %0"::"m" (last_task_used_math->tss.i387));
        }
// 现在，last_task_used_math 指向当前任务，以备当前任务被交换出去时使用。此时如果当前
// 任务用过协处理器，则恢复其状态。否则的话说明是第一次使用，于是就向协处理器发初始化
// 命令，并设置使用了协处理器标志。
        last_task_used_math=current;
        if (current->used_math) {
                __asm__("frstor %0"::"m" (current->tss.i387));
        } else {
                __asm__("fninit"::);            // 向协处理器发初始化命令。
                current->used_math=1;           // 设置使用已协处理器标志。
        }
}

```

```
109 /*
110  *  'schedule()' is the scheduler function.  This is GOOD CODE! There
111  * probably won't be any reason to change this, as it should work well
112  * in all circumstances (ie gives IO-bound processes good response etc).
113  * The one thing you might take a look at is the signal-handler code here.
114  *
115  *   NOTE!!  Task 0 is the 'idle' task, which gets called when no other
116  * tasks can run.  It can not be killed, and it cannot sleep. The 'state'
117  * information in task[0] is never used.
118  */
    /*
     * 'schedule()'是调度函数。这是个很好的代码！没有任何理由对它进行修改，因为
     * 它可以在所有的环境下工作（比如能够对 IO-边界处理很好的响应等）。只有一件
     * 事值得留意，那就是这里的信号处理代码。
     *
     *   注意！！任务 0 是个闲置('idle')任务，只有当没有其他任务可以运行时才调用
     * 它。它不能被杀死，也不能睡眠。任务 0 中的状态信息'state'是从来不用的。
     */
119 void schedule(void)
120 {
121         int i,next,c;
122         struct task_struct ** p;                // 任务结构指针的指针。
123
124 /* check alarm, wake up any interruptible tasks that have got a signal */
    /* 检测 alarm（进程的报警定时值），唤醒任何已得到信号的可中断任务 */
125
    // 从任务数组中最后一个任务开始循环检测 alarm。在循环时跳过空指针项。
126         for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
127                 if (*p) {
    // 如果设置过任务超时定时 timeout，并且已经超时，则复位超时定时值，并且如果任务处于可
    // 中断睡眠状态 TASK_INTERRUPTIBLE 下，将其置为就绪状态（TASK_RUNNING）。
128                         if ((*p)->timeout && (*p)->timeout < jiffies) {
129                                 (*p)->timeout = 0;
130                                 if ((*p)->state == TASK_INTERRUPTIBLE)
131                                         (*p)->state = TASK_RUNNING;
132                         }
    // 如果设置过任务的定时值 alarm，并且已经过期(alarm<jiffies),则在信号位图中置 SIGALRM
    // 信号，即向任务发送 SIGALARM 信号。然后清 alarm。该信号的默认操作是终止进程。jiffies
    // 是系统从开机开始算起的滴答数（10ms/滴答）。定义在 sched.h 第 139 行。
133                         if ((*p)->alarm && (*p)->alarm < jiffies) {
134                                 (*p)->signal |= (1<<(SIGALRM-1));
135                                 (*p)->alarm = 0;
136                         }
    // 如果信号位图中除被阻塞的信号外还有其他信号，并且任务处于可中断状态，则置任务为就绪
    // 状态。其中'~(_BLOCKABLE & (*p)->blocked)'用于忽略被阻塞的信号，但 SIGKILL 和 SIGSTOP
    // 不能被阻塞。
137                         if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
138                         (*p)->state==TASK_INTERRUPTIBLE)
139                                 (*p)->state=TASK_RUNNING;   //置为就绪（可执行）状态。
140                 }
141
142 /* this is the scheduler proper: */
    /* 这里是调度程序的主要部分 */
```

```
143
144         while (1) {
145                 c = -1;
146                 next = 0;
147                 i = NR_TASKS;
148                 p = &task[NR_TASKS];
    // 这段代码也是从任务数组的最后一个任务开始循环处理，并跳过不含任务的数组槽。比较每个
    // 就绪状态任务的 counter（任务运行时间的递减滴答计数）值，哪一个值大，运行时间还不长，
    // next 就指向哪个的任务号。
149                 while (--i) {
150                         if (!*--p)
151                                 continue;
152                         if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
153                                 c = (*p)->counter, next = i;
154                 }
    // 如果比较得出有 counter 值不等于 0 的结果，或者系统中没有一个可运行的任务存在（此时 c
    // 仍然为-1，next=0），则退出 144 行开始的循环，执行 161 行上的任务切换操作。否则就根据
    // 每个任务的优先权值，更新每一个任务的 counter 值，然后回到 125 行重新比较。counter 值
    // 的计算方式为 counter = counter /2 + priority。注意，这里计算过程不考虑进程的状态。
155                 if (c) break;
156                 for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
157                         if (*p)
158                                 (*p)->counter = ((*p)->counter >> 1) +
159                                         (*p)->priority;
160         }
    // 用下面宏（定义在 sched.h 中）把当前任务指针 current 指向任务号为 next 的任务，并切换
    // 到该任务中运行。在 146 行上 next 被初始化为 0。因此若系统中没有任何其他任务可运行时，
    // 则 next 始终为 0。因此调度函数会在系统空闲时去执行任务 0。 此时任务 0 仅执行 pause()
    // 系统调用，并又会调用本函数。
161         switch_to(next);                        // 切换到任务号为 next 的任务，并运行之。
162 }
163
    //// pause()系统调用。转换当前任务的状态为可中断的等待状态，并重新调度。
    // 该系统调用将导致进程进入睡眠状态，直到收到一个信号。该信号用于终止进程或者使进程
    // 调用一个信号捕获函数。只有当捕获了一个信号，并且信号捕获处理函数返回，pause()才
    // 会返回。此时 pause()返回值应该是 -1，并且 errno 被置为 EINTR。这里还没有完全实现
    // （直到 0.95 版）。
164 int sys_pause(void)
165 {
166         current->state = TASK_INTERRUPTIBLE;
167         schedule();
168         return 0;
169 }
170
    // 把当前任务置为指定的睡眠状态（可中断的或不可中断的），并让睡眠队列头指针指向当前任务。
    // 函数参数 p 是等待任务队列头指针。指针是含有一个变量地址的变量。这里参数 p 使用了指针的
    // 指针形式 '**p'，这是因为 C 函数参数只能传值，没有直接的方式让被调用函数改变调用该函数
    // 程序中变量的值。但是指针'*p'指向的目标（这里是任务结构）会改变，因此为了能修改调用该
    // 函数程序中原来就是指针变量的值，就需要传递指针'*p'的指针，即'**p'。参见程序前示例图中
    // p 指针的使用情况。
    // 参数 state 是任务睡眠使用的状态：TASK_UNINTERRUPTIBLE 或 TASK_INTERRUPTIBLE。处于不可
    // 中断睡眠状态（TASK_UNINTERRUPTIBLE）的任务需要内核程序利用 wake_up()函数明确唤醒之。
    // 处于可中断睡眠状态（TASK_INTERRUPTIBLE）可以通过信号、任务超时等手段唤醒 （置为就绪
```

```
    // 状态 TASK_RUNNING）。
    // *** 注意，由于本内核代码不是很成熟，因此下列与睡眠相关的代码存在一些问题，不宜深究。
171 static inline void __sleep_on(struct task_struct **p, int state)
172 {
173         struct task_struct *tmp;
174
    // 若指针无效，则退出。（指针所指的对象可以是 NULL，但指针本身不会为 0）。
    // 如果当前任务是任务 0，则死机(impossible!)。
175         if (!p)
176                 return;
177         if (current == &(init_task.task))
178                 panic("task[0] trying to sleep");
    // 让 tmp 指向已经在等待队列上的任务(如果有的话)，例如 inode->i_wait。并且将睡眠队列头
    // 的等待指针指向当前任务。这样就把当前任务插入到了 *p 的等待队列中。然后将当前任务置
    // 为指定的等待状态，并执行重新调度。
179         tmp = *p;
180         *p = current;
181         current->state = state;
182 repeat: schedule();
    // 只有当这个等待任务被唤醒时，程序才又会返回到这里，表示进程已被明确地唤醒并执行。
    // 如果等待队列中还有等待任务，并且队列头指针 *p 所指向的任务不是当前任务时，说明
    // 在本任务插入等待队列后还有任务进入等待队列。于是我们应该也要唤醒这个任务，而我
    // 们自己应按顺序让这些后面进入队列的任务唤醒，因此这里将等待队列头所指任务先置为
    // 就绪状态，而自己则置为不可中断等待状态，即自己要等待这些后续进队列的任务被唤醒
    // 而执行时来唤醒本任务。然后重新执行调度程序。
183         if (*p && *p != current) {
184                 (**p).state = 0;
185                 current->state = TASK_UNINTERRUPTIBLE;
186                 goto repeat;
187         }
    // 执行到这里，说明本任务真正被唤醒执行。此时等待队列头指针应该指向本任务，若它为
    // 空，则表明调度有问题，于是显示警告信息。最后我们让头指针指向在我们前面进入队列
    // 的任务（*p = tmp）。 若确实存在这样一个任务，即队列中还有任务（tmp 不为空），就
    // 唤醒之。最先进入队列的任务在唤醒后运行时最终会把等待队列头指针置成 NULL。
188         if (!*p)
189                 printk("Warning: *P = NULL\n\r");
190         if (*p = tmp)
191                 tmp->state=0;
192 }
193
    // 将当前任务置为可中断的等待状态（TASK_INTERRUPTIBLE），并放入头指针*p 指定的等待
    // 队列中。
194 void interruptible_sleep_on(struct task_struct **p)
195 {
196         __sleep_on(p,TASK_INTERRUPTIBLE);
197 }
198
    // 把当前任务置为不可中断的等待状态（TASK_UNINTERRUPTIBLE），并让睡眠队列头指针指向
    // 当前任务。只有明确地唤醒时才会返回。该函数提供了进程与中断处理程序之间的同步机制。
199 void sleep_on(struct task_struct **p)
200 {
201         __sleep_on(p,TASK_UNINTERRUPTIBLE);
202 }
```

```
203
    // 唤醒 *p 指向的任务。*p 是任务等待队列头指针。由于新等待任务是插入在等待队列头指针
    // 处的，因此唤醒的是最后进入等待队列的任务。若该任务已经处于停止或僵死状态，则显示
    // 警告信息。
204 void wake_up(struct task_struct **p)
205 {
206         if (p && *p) {
207                 if ((**p).state == TASK_STOPPED)          // 处于停止状态。
208                         printk("wake_up: TASK_STOPPED");
209                 if ((**p).state == TASK_ZOMBIE)           // 处于僵死状态。
210                         printk("wake_up: TASK_ZOMBIE");
211                 (**p).state=0;                            // 置为就绪状态 TASK_RUNNING。
212         }
213 }
214
215 /*
216  * OK, here are some floppy things that shouldn't be in the kernel
217  * proper. They are here because the floppy needs a timer, and this
218  * was the easiest way of doing it.
219  */
    /*
     * 好了，从这里开始是一些有关软盘的子程序，本不应该放在内核的主要部分
     * 中的。将它们放在这里是因为软驱需要定时处理，而放在这里是最方便的。
     */
    // 下面 220 -- 281 行代码用于处理软驱定时。在阅读这段代码之前请先看一下块设备一章中
    // 有关软盘驱动程序（floppy.c）后面的说明， 或者到阅读软盘块设备驱动程序时在来看这
    // 段代码。其中时间单位：1 个滴答 = 1/100 秒。
    // 下面数组 wait_motor[]用于存放等待软驱马达启动到正常转速的进程指针。数组索引 0-3
    // 分别对应软驱 A--D。数组 mon_timer[]存放各软驱马达启动所需要的滴答数。程序中默认
    // 启动时间为 50 个滴答（0.5 秒）。数组 moff_timer[] 存放各软驱在马达停转之前需维持
    // 的时间。程序中设定为 10000 个滴答（100 秒）。
220 static struct task_struct * wait_motor[4] = {NULL,NULL,NULL,NULL};
221 static int  mon_timer[4]={0,0,0,0};
222 static int moff_timer[4]={0,0,0,0};

    // 下面变量对应软驱控制器中当前数字输出寄存器。该寄存器每位的定义如下：
    // 位 7-4：分别控制驱动器 D-A 马达的启动。1 - 启动；0 - 关闭。
    // 位 3  ：1 - 允许 DMA 和中断请求；0 - 禁止 DMA 和中断请求。
    // 位 2  ：1 - 启动软盘控制器；     0 - 复位软盘控制器。
    // 位 1-0：00 - 11，用于选择控制的软驱 A-D。
    // 这里设置初值为：允许 DMA 和中断请求、启动 FDC。
223 unsigned char current_DOR = 0x0C;
224
    // 指定软驱启动到正常运转状态所需等待时间。
    // 参数 nr -- 软驱号(0--3)，返回值为滴答数。
    // 局部变量 selected 是选中软驱标志（blk_drv/floppy.c，123 行）。mask 是所选软驱对应的
    // 数字输出寄存器中启动马达比特位。mask 高 4 位是各软驱启动马达标志。
225 int ticks_to_floppy_on(unsigned int nr)
226 {
227         extern unsigned char selected;
228         unsigned char mask = 0x10 << nr;
229
    // 系统最多有 4 个软驱。首先预先设置好指定软驱 nr 停转之前需要经过的时间（100 秒）。然后
```

```
    // 取当前 DOR 寄存器值到临时变量 mask 中，并把指定软驱的马达启动标志置位。
230         if (nr>3)
231                 panic("floppy_on: nr>3");
232         moff_timer[nr]=10000;           /* 100 s = very big :-) */  // 停转维持时间。
233         cli();                          /* use floppy_off to turn it off */ // 关中断。
234         mask |= current_DOR;
    // 如果当前没有选择软驱，则首先复位其他软驱的选择位，然后置指定软驱选择位。
235         if (!selected) {
236                 mask &= 0xFC;
237                 mask |= nr;
238         }
    // 如果数字输出寄存器的当前值与要求的值不同，则向 FDC 数字输出端口输出新值(mask)，并且
    // 如果要求启动的马达还没有启动，则置相应软驱的马达启动定时器值（HZ/2 = 0.5 秒或 50 个
    // 滴答）。若已经启动，则再设置启动定时为 2 个滴答，能满足下面 do_floppy_timer()中先递
    // 减后判断的要求。执行本次定时代码的要求即可。此后更新当前数字输出寄存器 current_DOR。
239         if (mask != current_DOR) {
240                 outb(mask,FD_DOR);
241                 if ((mask ^ current_DOR) & 0xf0)
242                         mon_timer[nr] = HZ/2;
243                 else if (mon_timer[nr] < 2)
244                         mon_timer[nr] = 2;
245                 current_DOR = mask;
246         }
247         sti();                          // 开中断。
248         return mon_timer[nr];           // 最后返回启动马达所需的时间值。
249 }
250
    // 等待指定软驱马达启动所需的一段时间，然后返回。
    // 设置指定软驱的马达启动到正常转速所需的延时，然后睡眠等待。在定时中断过程中会一直
    // 递减判断这里设定的延时值。当延时到期，就会唤醒这里的等待进程。
251 void floppy_on(unsigned int nr)
252 {
    // 关中断。如果马达启动定时还没到，就一直把当前进程置为不可中断睡眠状态并放入等待马达
    // 运行的队列中。然后开中断。
253         cli();
254         while (ticks_to_floppy_on(nr))
255                 sleep_on(nr+wait_motor);
256         sti();
257 }
258
    // 置关闭相应软驱马达停转定时器（3 秒）。
    // 若不使用该函数明确关闭指定的软驱马达，则在马达开启 100 秒之后也会被关闭。
259 void floppy_off(unsigned int nr)
260 {
261         moff_timer[nr]=3*HZ;
262 }
263
    // 软盘定时处理子程序。更新马达启动定时值和马达关闭停转计时值。该子程序会在时钟定时
    // 中断过程中被调用，因此系统每经过一个滴答(10ms)就会被调用一次，随时更新马达开启或
    // 停转定时器的值。如果某一个马达停转定时到，则将数字输出寄存器马达启动位复位。
264 void do_floppy_timer(void)
265 {
266         int i;
```

```
267         unsigned char mask = 0x10;
268
269         for (i=0 ; i<4 ; i++,mask <<= 1) {
270                 if (!(mask & current_DOR))              // 如果不是 DOR 指定的马达则跳过。
271                         continue;
272                 if (mon_timer[i]) {                     // 如果马达启动定时到则唤醒进程。
273                         if (!--mon_timer[i])
274                                 wake_up(i+wait_motor);
275                 } else if (!moff_timer[i]) {            // 如果马达停转定时到则
276                         current_DOR &= ~mask;           // 复位相应马达启动位，并且
277                         outb(current_DOR,FD_DOR);       // 更新数字输出寄存器。
278                 } else
279                         moff_timer[i]--;                // 否则马达停转计时递减。
280         }
281 }
282
    // 下面是关于定时器的代码。最多可有 64 个定时器。
283 #define TIME_REQUESTS 64
284
    // 定时器链表结构和定时器数组。该定时器链表专用于供软驱关闭马达和启动马达定时操作。
    // 这种类型定时器类似现代 Linux 系统中的动态定时器（Dynamic Timer），仅供内核使用。
285 static struct timer_list {
286         long jiffies;                   // 定时滴答数。
287         void (*fn)();                   // 定时处理程序。
288         struct timer_list * next;       // 链接指向下一个定时器。
289 } timer_list[TIME_REQUESTS], * next_timer = NULL;  // next_timer 是定时器队列头指针。
290
    // 添加定时器。输入参数为指定的定时值(滴答数)和相应的处理程序指针。
    // 软盘驱动程序（floppy.c）利用该函数执行启动或关闭马达的延时操作。
    // 参数 jiffies - 以 10 毫秒计的滴答数；*fn()- 定时时间到时执行的函数。
291 void add_timer(long jiffies, void (*fn)(void))
292 {
293         struct timer_list * p;
294
    // 如果定时处理程序指针为空，则退出。否则关中断。
295         if (!fn)
296                 return;
297         cli();
    // 如果定时值<=0，则立刻调用其处理程序。并且该定时器不加入链表中。
298         if (jiffies <= 0)
299                 (fn)();
300         else {
    // 否则从定时器数组中，找一个空闲项。
301                 for (p = timer_list ; p < timer_list + TIME_REQUESTS ; p++)
302                         if (!p->fn)
303                                 break;
    // 如果已经用完了定时器数组，则系统崩溃☺。否则向定时器数据结构填入相应信息，并链入
    // 链表头。
304                 if (p >= timer_list + TIME_REQUESTS)
305                         panic("No more time requests free");
306                 p->fn = fn;
307                 p->jiffies = jiffies;
308                 p->next = next_timer;
```

```
309                 next_timer = p;
    // 链表项按定时值从小到大排序。在排序时减去排在前面需要的滴答数，这样在处理定时器时
    // 只要查看链表头的第一项的定时是否到期即可。[[?? 这段程序好象没有考虑周全。如果新
    // 插入的定时器值小于原来头一个定时器值时则根本不会进入循环中，但此时还是应该将紧随
    // 其后面的一个定时器值减去新的第 1 个的定时值。即如果第 1 个定时值<=第 2 个，则第 2 个
    // 定时值扣除第 1 个的值即可，否则进入下面循环中进行处理。]]
310                 while (p->next && p->next->jiffies < p->jiffies) {
311                         p->jiffies -= p->next->jiffies;
312                         fn = p->fn;
313                         p->fn = p->next->fn;
314                         p->next->fn = fn;
315                         jiffies = p->jiffies;
316                         p->jiffies = p->next->jiffies;
317                         p->next->jiffies = jiffies;
318                         p = p->next;
319                 }
320         }
321         sti();
322 }
323
    //// 时钟中断 C 函数处理程序，在 sys_call.s 中的_timer_interrupt（189 行）被调用。
    // 参数 cpl 是当前特权级 0 或 3，是时钟中断发生时正被执行的代码选择符中的特权级。
    // cpl=0 时表示中断发生时正在执行内核代码；cpl=3 时表示中断发生时正在执行用户代码。
    // 对于一个进程由于执行时间片用完时，则进行任务切换。并执行一个计时更新工作。
324 void do_timer(long cpl)
325 {
326         static int blanked = 0;
327
    // 首先判断是否经过了一定时间而让屏幕黑屏（blankout）。如果 blankcount 计数不为零，
    // 或者黑屏延时间隔时间 blankinterval 为 0 的话，那么若已经处于黑屏状态（黑屏标志
    // blanked = 1），则让屏幕恢复显示。若 blankcount 计数不为零，则递减之，并且复位
    // 黑屏标志。
328         if (blankcount || !blankinterval) {
329                 if (blanked)
330                         unblank_screen();
331                 if (blankcount)
332                         blankcount--;
333                 blanked = 0;
    // 否则的话若黑屏标志未置位，则让屏幕黑屏，并且设置黑屏标志。
334         } else if (!blanked) {
335                 blank_screen();
336                 blanked = 1;
337         }
    // 接着处理硬盘操作超时问题。如果硬盘超时计数递减之后为 0，则进行硬盘访问超时处理。
338         if (hd_timeout)
339                 if (!--hd_timeout)
340                         hd_times_out();  // 硬盘访问超时处理（blk_drv/hdc，318 行）。
341
    // 如果发声计数次数到，则关闭发声。(向 0x61 口发送命令，复位位 0 和 1。位 0 控制 8253
    // 计数器 2 的工作，位 1 控制扬声器)。
342         if (beepcount)          // 扬声器发声时间滴答数（chr_drv/console.c,950 行）。
343                 if (!--beepcount)
344                         sysbeepstop();
```

```
345
    // 如果当前特权级(cpl)为 0（最高，表示是内核程序在工作），则将内核代码运行时间 stime
    // 递增；[ Linus 把内核程序统称为超级用户(supervisor)的程序，见 sys_call.s，207 行
    // 上的英文注释。这种称呼来自于 Intel CPU 手册。] 如果 cpl > 0，则表示是一般用户程序
    // 在工作，增加 utime。
346         if (cpl)
347                 current->utime++;
348         else
349                 current->stime++;
350
    // 如果有定时器存在，则将链表第 1 个定时器的值减 1。如果已等于 0，则调用相应的处理程序，
    // 并将该处理程序指针置为空。然后去掉该项定时器。next_timer 是定时器链表的头指针。
351         if (next_timer) {
352                 next_timer->jiffies--;
353                 while (next_timer && next_timer->jiffies <= 0) {
354                         void (*fn)(void);       // 这里插入了一个函数指针定义！！☹
355
356                         fn = next_timer->fn;
357                         next_timer->fn = NULL;
358                         next_timer = next_timer->next;
359                         (fn)();                 // 调用定时处理函数。
360                 }
361         }
    // 如果当前软盘控制器 FDC 的数字输出寄存器中马达启动位有置位的，则执行软盘定时程序。
362         if (current_DOR & 0xf0)
363                 do_floppy_timer();
    // 如果进程运行时间还没完，则退出。否则置当前任务运行计数值为 0。并且若发生时钟中断时
    // 正在内核代码中运行则返回，否则调用执行调度函数。
364         if ((--current->counter)>0) return;
365         current->counter=0;
366         if (!cpl) return;               // 对于内核态程序，不依赖 counter 值进行调度。
367         schedule();
368 }
369
    // 系统调用功能 - 设置报警定时时间值(秒)。
    // 若参数 seconds 大于 0，则设置新定时值，并返回原定时时刻还剩余的间隔时间。否则返回 0。
    // 进程数据结构中报警定时值 alarm 的单位是系统滴答（1 滴答为 10 毫秒），它是系统开机起到
    // 设置定时操作时系统滴答值 jiffies 和转换成滴答单位的定时值之和，即'jiffies + HZ*定时
    // 秒值'。而参数给出的是以秒为单位的定时值，因此本函数的主要操作是进行两种单位的转换。
    // 其中常数 HZ = 100，是内核系统运行频率。定义在 include/sched.h 第 4 行上。
    // 参数 seconds 是新的定时时间值，单位是秒。
370 int sys_alarm(long seconds)
371 {
372         int old = current->alarm;
373
374         if (old)
375                 old = (old - jiffies) / HZ;
376         current->alarm = (seconds>0)?(jiffies+HZ*seconds):0;
377         return (old);
378 }
379
    // 取当前进程号 pid。
380 int sys_getpid(void)
```

```
{
        return current->pid;
}

// 取父进程号 ppid。
int sys_getppid(void)
{
        return current->p_pptr->pid;
}

// 取用户号 uid。
int sys_getuid(void)
{
        return current->uid;
}

// 取有效的用户号 euid。
int sys_geteuid(void)
{
        return current->euid;
}

// 取组号 gid。
int sys_getgid(void)
{
        return current->gid;
}

// 取有效的组号 egid。
int sys_getegid(void)
{
        return current->egid;
}

// 系统调用功能 -- 降低对 CPU 的使用优先权（有人会用吗？☺）。
// 应该限制 increment 为大于 0 的值，否则可使优先权增大！！
int sys_nice(long increment)
{
        if (current->priority-increment>0)
                current->priority -= increment;
        return 0;
}

// 内核调度程序的初始化子程序。
void sched_init(void)
{
        int i;
        struct desc_struct * p;         // 描述符表结构指针。

// Linux 系统开发之初，内核不成熟。内核代码会被经常修改。Linus 怕自己无意中修改了这些
// 关键性的数据结构，造成与 POSIX 标准的不兼容。这里加入下面这个判断语句并无必要，纯粹
// 是为了提醒自己以及其他修改内核代码的人。
        if (sizeof(struct sigaction) != 16)     // sigaction 是存放有关信号状态的结构。
```

```
423                 panic("Struct sigaction MUST be 16 bytes");
    // 在全局描述符表中设置初始任务（任务 0）的任务状态段描述符和局部数据表描述符。
    // FIRST_TSS_ENTRY 和 FIRST_LDT_ENTRY 的值分别是 4 和 5，定义在 include/linux/sched.h
    // 中；gdt 是一个描述符表数组（include/linux/head.h ），实际上对应程序 head.s 中
    // 第 234 行上的全局描述符表基址（_gdt）。因此 gdt + FIRST_TSS_ENTRY 即为
    // gdt[FIRST_TSS_ENTRY]（即是 gdt[4]），也即 gdt 数组第 4 项的地址。 参见
    // include/asm/system.h,第 65 行开始。
424         set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss));
425         set_ldt_desc(gdt+FIRST_LDT_ENTRY,&(init_task.task.ldt));
    // 清任务数组和描述符表项（注意 i=1 开始，所以初始任务的描述符还在）。描述符项结构
    // 定义在文件 include/linux/head.h 中。
426         p = gdt+2+FIRST_TSS_ENTRY;
427         for(i=1;i<NR_TASKS;i++) {
428                 task[i] = NULL;
429                 p->a=p->b=0;
430                 p++;
431                 p->a=p->b=0;
432                 p++;
433         }
434 /* Clear NT, so that we won't have troubles with that later on */
    /* 清除标志寄存器中的位 NT，这样以后就不会有麻烦 */
    // EFLAGS 中的 NT 标志位用于控制任务的嵌套调用。当 NT 位置位时，那么当前中断任务执行
    // IRET 指令时就会引起任务切换。NT 指出 TSS 中的 back_link 字段是否有效。NT=0 时无效。
435         __asm__("pushfl ; andl $0xffffbfff,(%esp) ; popfl");     // 复位 NT 标志。
    // 将任务 0 的 TSS 段选择符加载到任务寄存器 tr。将局部描述符表段选择符加载到局部描述
    // 符表寄存器 ldtr 中。注意！！是将 GDT 中相应 LDT 描述符的选择符加载到 ldtr。只明确加
    // 这一次，以后新任务 LDT 的加载，是 CPU 根据 TSS 中的 LDT 项自动加载。
436         ltr(0);                     // 定义在 include/linux/sched.h 第 157-158 行。
437         lldt(0);                    // 其中参数（0）是任务号。
    // 下面代码用于初始化 8253 定时器。通道 0，选择工作方式 3，二进制计数方式。通道 0 的
    // 输出引脚接在中断控制主芯片的 IRQ0 上，它每 10 毫秒发出一个 IRQ0 请求。LATCH 是初始
    // 定时计数值。
438         outb_p(0x36,0x43);              /* binary, mode 3, LSB/MSB, ch 0 */
439         outb_p(LATCH & 0xff , 0x40);    /* LSB */    // 定时值低字节。
440         outb(LATCH >> 8 , 0x40);        /* MSB */    // 定时值高字节。
    // 设置时钟中断处理程序句柄（设置时钟中断门）。修改中断控制器屏蔽码，允许时钟中断。
    // 然后设置系统调用中断门。这两个设置中断描述符表 IDT 中描述符的宏定义在文件
    // include/asm/system.h 中第 33、39 行处。两者的区别参见 system.h 文件开始处的说明。
441         set_intr_gate(0x20,&timer_interrupt);
442         outb(inb_p(0x21)&~0x01,0x21);
443         set_system_gate(0x80,&system_call);
444 }
445
```