程序 14-26 linux/include/linux/sched.h

```
1  #ifndef _SCHED_H
2  #define _SCHED_H
3
4  #define HZ 100                          // 定义系统时钟滴答频率(1 百赫兹，每个滴答 10ms)
5
6  #define NR_TASKS        64              // 系统中同时最多任务（进程）数。
7  #define TASK_SIZE       0x04000000      // 每个任务的长度（64MB）。
8  #define LIBRARY_SIZE    0x00400000      // 动态加载库长度（4MB）。
9
10 #if (TASK_SIZE & 0x3fffff)
11 #error "TASK_SIZE must be multiple of 4M"      // 任务长度必须是 4MB 的倍数。
12 #endif
13
14 #if (LIBRARY_SIZE & 0x3fffff)
15 #error "LIBRARY_SIZE must be a multiple of 4M" // 库长度也必须是 4MB 的倍数。
16 #endif
17
18 #if (LIBRARY_SIZE >= (TASK_SIZE/2))
19 #error "LIBRARY_SIZE too damn big!"            // 加载库的长度不得大于任务长度的一半。
20 #endif
21
22 #if (((TASK_SIZE>>16)*NR_TASKS) != 0x10000)
23 #error "TASK_SIZE*NR_TASKS must be 4GB"        // 任务长度*任务总个数必须为 4GB。
24 #endif
25
   // 在进程逻辑地址空间中动态库被加载的位置（60MB 处）。
26 #define LIBRARY_OFFSET (TASK_SIZE - LIBRARY_SIZE)
27
   // 下面宏 CT_TO_SECS 和 CT_TO_USECS 用于把系统当前嘀嗒数转换成用秒值加微秒值表示。
28 #define CT_TO_SECS(x)   ((x) / HZ)
29 #define CT_TO_USECS(x)  (((x) % HZ) * 1000000/HZ)
30
31 #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。
32 #define LAST_TASK task[NR_TASKS-1]      // 任务数组中的最后一项任务。
33
34 #include <linux/head.h>
35 #include <linux/fs.h>
36 #include <linux/mm.h>
37 #include <sys/param.h>
38 #include <sys/time.h>
39 #include <sys/resource.h>
40 #include <signal.h>
41
42 #if (NR_OPEN > 32)
43 #error "Currently the close-on-exec-flags and select masks are in one long, max 32 files/proc"
44 #endif
45
   // 这里定义了进程运行时可能处的状态。
46 #define TASK_RUNNING            0   // 进程正在运行或已准备就绪。
47 #define TASK_INTERRUPTIBLE      1   // 进程处于可中断等待状态。
48 #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。
49 #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。
```

```
50 #define TASK_STOPPED            4   // 进程已停止。
51
52 #ifndef NULL
53 #define NULL ((void *) 0)           // 定义 NULL 为空指针。
54 #endif
55
   // 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 )
56 extern int copy_page_tables(unsigned long from, unsigned long to, long size);
   // 释放页表所指定的内存块及页表本身。( mm/memory.c, 150 )
57 extern int free_page_tables(unsigned long from, unsigned long size);
58
   // 调度程序的初始化函数。( kernel/sched.c, 385 )
59 extern void sched_init(void);
   // 进程调度函数。( kernel/sched.c, 104 )
60 extern void schedule(void);
   // 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 )
61 extern void trap_init(void);
   // 显示内核出错信息，然后进入死循环。( kernel/panic.c, 16 )。
62 extern void panic(const char * str);
   // 往 tty 上写指定长度的字符串。( kernel/chr_drv/tty_io.c, 290 )。
63 extern int tty_write(unsigned minor,char * buf,int count);
64
65 typedef int (*fn_ptr)();                    // 定义函数指针类型。
66
   // 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。
67 struct i387_struct {
68         long    cwd;            // 控制字(Control word)。
69         long    swd;            // 状态字(Status word)。
70         long    twd;            // 标记字(Tag word)。
71         long    fip;            // 协处理器代码指针。
72         long    fcs;            // 协处理器代码段寄存器。
73         long    foo;            // 内存操作数的偏移位置。
74         long    fos;            // 内存操作数的段值。
75         long    st_space[20];   /* 8*10 bytes for each FP-reg = 80 bytes */
76 };                              /* 8 个 10 字节的协处理器累加器。*/
77
   // 任务状态段数据结构。
78 struct tss_struct {
79         long    back_link;      /* 16 high bits zero */
80         long    esp0;
81         long    ss0;            /* 16 high bits zero */
82         long    esp1;
83         long    ss1;            /* 16 high bits zero */
84         long    esp2;
85         long    ss2;            /* 16 high bits zero */
86         long    cr3;
87         long    eip;
88         long    eflags;
89         long    eax,ecx,edx,ebx;
90         long    esp;
91         long    ebp;
92         long    esi;
93         long    edi;
```

```
 94         long    es;             /* 16 high bits zero */
 95         long    cs;             /* 16 high bits zero */
 96         long    ss;             /* 16 high bits zero */
 97         long    ds;             /* 16 high bits zero */
 98         long    fs;             /* 16 high bits zero */
 99         long    gs;             /* 16 high bits zero */
100         long    ldt;            /* 16 high bits zero */
101         long    trace_bitmap;   /* bits: trace 0, bitmap 16-31 */
102         struct i387_struct i387;
103 };
104
    // 下面是任务（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                优先数。任务开始运行时 counter=priority，越大运行越长。
    // long signal                  信号位图，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       代码长度 + 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 pgrp                    进程组号。
    // long session                 会话号。
    // long leader                  会话首领。
    // int groups[NGROUPS]          进程所属组号。一个进程可属于多个组。
    // task_struct *p_pptr          指向父进程的指针。
    // task_struct *p_cptr          指向最新子进程的指针。
    // task_struct *p_ysptr         指向比自己后创建的相邻进程的指针。
    // task_struct *p_osptr         指向比自己早创建的相邻进程的指针。
    // 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 timeout                 内核定时超时值。
    // 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。
    // struct rlimit rlim[RLIM_NLIMITS]  进程资源使用统计数组。
    // unsigned int flags;          各进程的标志，在下面第 149 行开始定义（还未使用）。
    // unsigned short used_math     标志：是否使用了协处理器。
    // ------------------------
    // int tty                      进程使用 tty 终端的子设备号。-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 节点结构指针。
    // struct m_inode * root         根目录 i 节点结构指针。
    // struct m_inode * executable   执行文件 i 节点结构指针。
    // struct m_inode * library      被加载库文件 i 节点结构指针。
    // 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]   文件结构指针表，最多 32 项。表项号即是文件描述符的值。
    // struct desc_struct ldt[3]     局部描述符表。0-空，1-代码段 cs，2-数据和堆栈段 ds&ss。
    // struct tss_struct tss         进程的任务状态段信息结构。
    // ======================================
105 struct task_struct {
106 /* these are hardcoded - don't touch */
107         long state;     /* -1 unrunnable, 0 runnable, >0 stopped */
108         long counter;
109         long priority;
110         long signal;
111         struct sigaction sigaction[32];
112         long blocked;   /* bitmap of masked signals */
113 /* various fields */
114         int exit_code;
115         unsigned long start_code,end_code,end_data,brk,start_stack;
116         long pid,pgrp,session,leader;
117         int     groups[NGROUPS];
118         /*
119          * pointers to parent process, youngest child, younger sibling,
120          * older sibling, respectively.  (p->father can be replaced with
121          * p->p_pptr->pid)
122          */
123         struct task_struct      *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
124         unsigned short uid,euid,suid;
125         unsigned short gid,egid,sgid;
126         unsigned long timeout,alarm;
127         long utime,stime,cutime,cstime,start_time;
128         struct rlimit rlim[RLIM_NLIMITS];
129         unsigned int flags;     /* per process flags, defined below */
130         unsigned short used_math;
131 /* file system info */
132         int tty;                /* -1 if no tty, so it must be signed */
133         unsigned short umask;
134         struct m_inode * pwd;
135         struct m_inode * root;
136         struct m_inode * executable;
137         struct m_inode * library;
138         unsigned long close_on_exec;
139         struct file * filp[NR_OPEN];
140 /* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
141         struct desc_struct ldt[3];
142 /* tss for this task */
143         struct tss_struct tss;
144 };
145
146 /*
147  * Per process flags
148  */
```

```
    /* 每个进程的标志 */  /* 打印对齐警告信息。还未实现，仅用于 486 */
149 #define PF_ALIGNWARN    0x00000001      /* Print alignment warning msgs */
150                                         /* Not implemented yet, only for 486*/
151
152 /*
153  *  INIT_TASK is used to set up the first task table, touch at
154  * your own risk!. Base=0, limit=0x9ffff (=640kB)
155  */
    /*
     * INIT_TASK 用于设置第 1 个任务表，若想修改，责任自负☺!
     * 基址 Base = 0，段长 limit = 0x9ffff（=640kB）。
     */
    // 对应上面任务结构的第 1 个任务的信息。
156 #define INIT_TASK \
157 /* state etc */ { 0,15,15, \      // state, counter, priority
158 /* signals */   0,{{},},0, \      // signal, sigaction[32], blocked
159 /* ec,brk... */ 0,0,0,0,0,0, \  // exit_code,start_code,end_code,end_data,brk,start_stack
160 /* pid etc.. */ 0,0,0,0, \        // pid, pgrp, session, leader
161 /* suppl grps*/ {NOGROUP,}, \     // groups[]
162 /* proc links*/ &init_task.task,0,0,0, \    // p_pptr, p_cptr, p_ysptr, p_osptr
163 /* uid etc */   0,0,0,0,0,0, \    // uid, euid, suid, gid, egid, sgid
164 /* timeout */   0,0,0,0,0,0,0,0, \  // alarm,utime,stime,cutime,cstime,start_time,used_math
165 /* rlimits */   { {0x7fffffff, 0x7fffffff}, {0x7fffffff, 0x7fffffff},  \
166                   {0x7fffffff, 0x7fffffff}, {0x7fffffff, 0x7fffffff}, \
167                   {0x7fffffff, 0x7fffffff}, {0x7fffffff, 0x7fffffff}}, \
168 /* flags */     0, \              // flags
169 /* math */      0, \              // used_math, tty,umask,pwd,root,executable,close_on_exec
170 /* fs info */   -1,0022,NULL,NULL,NULL,NULL,0, \
171 /* filp */      {NULL,}, \        // filp[20]
172         { \                       // ldt[3]
173                 {0,0}, \
174 /* ldt */       {0x9f,0xc0fa00}, \  // 代码长 640K，基址 0x0，G=1，D=1，DPL=3，P=1 TYPE=0xa
175                 {0x9f,0xc0f200}, \  // 数据长 640K，基址 0x0，G=1，D=1，DPL=3，P=1 TYPE=0x2
176         }, \
177 /*tss*/ {0,PAGE_SIZE+(long)&init_task,0x10,0,0,0,0,(long)&pg_dir,\       // tss
178          0,0,0,0,0,0,0,0, \
179          0,0,0x17,0x17,0x17,0x17,0x17,0x17, \
180          _LDT(0),0x80000000, \
181                 {} \
182         }, \
183 }
184
185 extern struct task_struct *task[NR_TASKS];      // 任务指针数组。
186 extern struct task_struct *last_task_used_math; // 上一个使用过协处理器的进程。
187 extern struct task_struct *current;             // 当前运行进程结构指针变量。
188 extern unsigned long volatile jiffies;          // 从开机开始算起的滴答数（10ms/滴答）。
189 extern unsigned long startup_time;              // 开机时间。从 1970:0:0:0 开始计时的秒数。
190 extern int jiffies_offset;                      // 用于累计需要调整的时间嘀嗒数。
191
192 #define CURRENT_TIME (startup_time+(jiffies+jiffies_offset)/HZ)  // 当前时间（秒数）。
193
    // 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 )
194 extern void add_timer(long jiffies, void (*fn)(void));
```

```
    // 不可中断的等待睡眠。( kernel/sched.c )
195 extern void sleep_on(struct task_struct ** p);
    // 可中断的等待睡眠。( kernel/sched.c )
196 extern void interruptible_sleep_on(struct task_struct ** p);
    // 明确唤醒睡眠的进程。( kernel/sched.c )
197 extern void wake_up(struct task_struct ** p);
    // 检查当前进程是否在指定的用户组 grp 中。
198 extern int in_group_p(gid_t grp);
199
200 /*
201  * Entry into gdt where to find first TSS. 0-nul, 1-cs, 2-ds, 3-syscall
202  * 4-TSS0, 5-LDT0, 6-TSS1 etc ...
203  */
    /*
     * 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。
     */
    // 从该英文注释可以猜想到，Linus 当时曾想把系统调用的代码专门放在 GDT 表中第 4 个独立的段中。
    // 但后来并没有那样做，于是就一直把 GDT 表中第 4 个描述符项（上面 syscall 项）闲置在一旁。
    // 下面定义宏：全局表中第 1 个任务状态段(TSS)描述符的选择符索引号。
204 #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。
205 #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 段描述符的选择符值（偏移量）。
    // 因每个描述符占 8 字节，因此 FIRST_TSS_ENTRY<<3 表示该描述符在 GDT 表中的起始偏移位置。
    // 因为每个任务使用 1 个 TSS 和 1 个 LDT 描述符，共占用 16 字节，因此需要 n<<4 来表示对应
    // TSS 起始位置。该宏得到的值正好也是该 TSS 的选择符值。
206 #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 段描述符的选择符值（偏移量）。
207 #define _LDT(n) ((((unsigned long) n)<<4)+(FIRST_LDT_ENTRY<<3))
    // 宏定义，把第 n 个任务的 TSS 段选择符加载到任务寄存器 TR 中。
208 #define ltr(n) __asm__("ltr %%ax"::"a" (_TSS(n)))
    // 宏定义，把第 n 个任务的 LDT 段选择符加载到局部描述符表寄存器 LDTR 中。
209 #define lldt(n) __asm__("lldt %%ax"::"a" (_LDT(n)))
    // 取当前运行任务的任务号（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 )。
210 #define str(n) \
211 __asm__("str %%ax\n\t" \                    // 将任务寄存器中 TSS 段的选择符复制到 ax 中。
212         "subl %2,%%eax\n\t" \               // (eax - FIRST_TSS_ENTRY*8)➔eax
213         "shrl $4,%%eax" \                   // (eax/16)➔eax = 当前任务号。
214         :"=a" (n) \
215         :"a" (0),"i" (FIRST_TSS_ENTRY<<3))
216 /*
217  *      switch_to(n) should switch tasks to task nr n, first
218  * checking that n isn't the current task, in which case it does nothing.
219  * This also clears the TS-flag if the task we switched to has used
220  * tha math co-processor latest.
221  */
    /*
     * 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。如果我们切换到的任务最近（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。
     */
```

```
    // 跳转到一个任务的 TSS 段选择符组成的地址处会造成 CPU 进行任务切换操作。
    // 输入：%0 - 指向__tmp；                %1 - 指向__tmp.b 处，用于存放新 TSS 的选择符；
    //       dx - 新任务 n 的 TSS 段选择符；   ecx - 新任务 n 的任务结构指针 task[n]。
    // 其中临时数据结构__tmp 用于组建 177 行远跳转（far jump）指令的操作数。该操作数由 4 字节
    // 偏移地址和 2 字节的段选择符组成。因此__tmp 中 a 的值是 32 位偏移值，而 b 的低 2 字节是新
    // TSS 段的选择符（高 2 字节不用）。跳转到 TSS 段选择符会造成任务切换到该 TSS 对应的进程。
    // 对于造成任务切换的长跳转，a 值无用。177 行上的内存间接跳转指令使用 6 字节操作数作为跳
    // 转目的地的长指针，其格式为：jmp 16 位段选择符：32 位偏移值。但在内存中操作数的表示顺
    // 序与这里正好相反。任务切换回来之后，在判断原任务上次执行是否使用过协处理器时，是通过
    // 将原任务指针与保存在 last_task_used_math 变量中的上次使用过协处理器任务指针进行比较而
    // 作出的，参见文件 kernel/sched.c 中有关 math_state_restore()函数的说明。
222 #define switch_to(n) {\
223 struct {long a,b;} __tmp; \
224 __asm__("cmpl %%ecx,_current\n\t" \      // 任务 n 是当前任务吗?(current ==task[n]?)
225         "je 1f\n\t" \                    // 是，则什么都不做，退出。
226         "movw %%dx,%1\n\t" \             // 将新任务 TSS 的 16 位选择符存入__tmp.b 中。
227         "xchgl %%ecx,_current\n\t" \     // current = task[n]；ecx = 被切换出的任务。
228         "ljmp %0\n\t" \                  // 执行长跳转至*&__tmp，造成任务切换。
                                             // 在任务切换回来后才会继续执行下面的语句。
229         "cmpl %%ecx,_last_task_used_math\n\t" \     // 原任务上次使用过协处理器吗？
230         "jne 1f\n\t" \                   // 没有则跳转，退出。
231         "clts\n" \                       // 原任务上次使用过协处理器，则清 cr0 中的任务
232         "1:" \                           // 切换标志 TS。
233         ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
234         "d" (_TSS(n)),"c" ((long) task[n])); \
235 }
236
    // 页面地址对准。（在内核代码中没有任何地方引用!!）
237 #define PAGE_ALIGN(n) (((n)+0xfff)&0xfffff000)
238
    // 设置位于地址 addr 处描述符中的各基地址字段(基地址是 base)。
    // %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。
239 #define _set_base(addr,base) \
240 __asm__("movw %%dx,%0\n\t" \             // 基址 base 低 16 位(位 15-0)➔[addr+2]。
241         "rorl $16,%%edx\n\t" \           // edx 中基址高 16 位(位 31-16)➔dx。
242         "movb %%dl,%1\n\t" \             // 基址高 16 位中的低 8 位(位 23-16)➔[addr+4]。
243         "movb %%dh,%2" \                 // 基址高 16 位中的高 8 位(位 31-24)➔[addr+7]。
244         ::"m" (*((addr)+2)), \
245           "m" (*((addr)+4)), \
246           "m" (*((addr)+7)), \
247           "d" (base) \
248         :"dx")                   // 告诉 gcc 编译器 edx 寄存器中的值已被嵌入汇编程序改变了。
249
    // 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。
250 #define _set_limit(addr,limit) \
251 __asm__("movw %%dx,%0\n\t" \             // 段长 limit 低 16 位(位 15-0)➔[addr]。
252         "rorl $16,%%edx\n\t" \           // edx 中的段长高 4 位(位 19-16)➔dl。
253         "movb %1,%%dh\n\t" \             // 取原[addr+6]字节➔dh，其中高 4 位是些标志。
254         "andb $0xf0,%%dh\n\t" \          // 清 dh 的低 4 位(将存放段长的位 19-16)。
255         "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，
256         "movb %%dl,%1" \                 // 并放会[addr+6]处。
257         ::"m" (*(addr)), \
```

```
258             "m" (*((addr)+6)), \
259             "d" (limit) \
260         :"dx")
261
    // 设置局部描述符表中 ldt 描述符的基地址字段。
262 #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。
263 #define set_limit(ldt,limit) _set_limit( ((char *)&(ldt)) , (limit-1)>>12 )
264
    // 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。
265 #define _get_base(addr) ({\
266 unsigned long __base; \
267 __asm__("movb %3,%%dh\n\t" \              // 取[addr+7]处基址高 16 位的高 8 位(位 31-24)➔dh。
268         "movb %2,%%dl\n\t" \              // 取[addr+4]处基址高 16 位的低 8 位(位 23-16)➔dl。
269         "shll $16,%%edx\n\t" \            // 基地址高 16 位移到 edx 中高 16 位处。
270         "movw %1,%%dx" \                  // 取[addr+2]处基址低 16 位(位 15-0)➔dx。
271         :"=d" (__base) \                  // 从而 edx 中含有 32 位的段基地址。
272         :"m" (*((addr)+2)), \
273          "m" (*((addr)+4)), \
274          "m" (*((addr)+7))); \
275 __base;})
276
    // 取局部描述符表中 ldt 所指段描述符中的基地址。
277 #define get_base(ldt) _get_base( ((char *)&(ldt)) )
278
    // 取段选择符 segment 指定的描述符中的段限长值。
    // 指令 lsl 是 Load Segment Limit 缩写。它从指定段描述符中取出分散的限长比特位拼成完整的
    // 段限长值放入指定寄存器中。所得的段限长是实际字节数减 1，因此这里还需要加 1 后才返回。
    // %0 - 存放段长值(字节数)；%1 - 段选择符 segment。
279 #define get_limit(segment) ({ \
280 unsigned long __limit; \
281 __asm__("lsll %1,%0\n\tincl %0":"=r" (__limit):"r" (segment)); \
282 __limit;})
283
284 #endif
285
```