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

/*
 *  'fork.c' contains the help-routines for the 'fork' system call
 * (see also system_call.s), and some misc functions ('verify_area').
 * Fork is rather simple, once you get the hang of it, but the memory
 * management can be a bitch. See 'mm/mm.c': 'copy_page_tables()'
 */
/*
 * 'fork.c'中含有系统调用'fork'的辅助子程序（参见 system_call.s），以及一些
 * 其他函数（'verify_area'）。一旦你了解了 fork，就会发现它是非常简单的，但
 * 内存管理却有些难度。参见'mm/memory.c'中的'copy_page_tables()'函数。
 */
#include <errno.h>          // 错误号头文件。包含系统中各种出错号。

#include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 的数据。
#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 <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 <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。

// 写页面验证。若页面不可写，则复制页面。定义在 mm/memory.c 第 261 行开始。
extern void write_verify(unsigned long address);

long last_pid=0;            // 最新进程号，其值会由 get_empty_process()生成。

//// 进程空间区域写前验证函数。
// 对于 80386 CPU，在执行特权级 0 代码时不会理会用户空间中的页面是否是页保护的，因此
// 在执行内核代码时用户空间中数据页面保护标志起不了作用，写时复制机制也就失去了作用。
// verify_area()函数就用于此目的。但对于 80486 或后来的 CPU，其控制寄存器 CR0 中有一个
// 写保护标志 WP（位 16），内核可以通过设置该标志来禁止特权级 0 的代码向用户空间只读
// 页面执行写数据，否则将导致发生写保护异常。从而 486 以上 CPU 可以通过设置该标志来达
// 到使用本函数同样的目的。
// 该函数对当前进程逻辑地址从 addr 到 addr + size 这一段范围以页为单位执行写操作前
// 的检测操作。由于检测判断是以页面为单位进行操作，因此程序首先需要找出 addr 所在页
// 面开始地址 start，然后 start 加上进程数据段基址，使这个 start 变换成 CPU 4G 线性空
// 间中的地址。最后循环调用 write_verify() 对指定大小的内存空间进行写前验证。若页面
// 是只读的，则执行共享检验和复制页面操作（写时复制）。
void verify_area(void * addr,int size)
{
        unsigned long start;

// 首先将起始地址 start 调整为其所在页的左边界开始位置，同时相应地调整验证区域大小。
// 下句中的 start & 0xfff 用来获得指定起始位置 addr（也即 start）在所在页面中的偏移
// 值，原验证范围 size 加上这个偏移值即扩展成以 addr 所在页面起始位置开始的范围值。
// 因此在 30 行上 也需要把验证开始位置 start 调整成页面边界值。参见前面的图“内存验
// 证范围的调整”。
        start = (unsigned long) addr;
        size += start & 0xfff;
```

```
30          start &= 0xfffff000;                    // 此时 start 是当前进程空间中的逻辑地址。
    // 下面把 start 加上进程数据段在线性地址空间中的起始基址，变成系统整个线性空间中的地
    // 址位置。对于 Linux 0.1x 内核，其数据段和代码段在线性地址空间中的基址和限长均相同。
    // 然后循环进行写页面验证。若页面不可写，则复制页面。（mm/memory.c，274 行）
31          start += get_base(current->ldt[2]);     // include/linux/sched.h，277 行。
32          while (size>0) {
33                  size -= 4096;
34                  write_verify(start);
35                  start += 4096;
36          }
37  }
38
    // 复制内存页表。
    // 参数 nr 是新任务号；p 是新任务数据结构指针。该函数为新任务在线性地址空间中设置代码
    // 段和数据段基址、限长，并复制页表。  由于 Linux 系统采用了写时复制（copy on write）
    // 技术， 因此这里仅为新进程设置自己的页目录表项和页表项，而没有实际为新进程分配物理
    // 内存页面。此时新进程与其父进程共享所有内存页面。操作成功返回 0，否则返回出错号。
39  int copy_mem(int nr,struct task_struct * p)
40  {
41          unsigned long old_data_base,new_data_base,data_limit;
42          unsigned long old_code_base,new_code_base,code_limit;
43
    // 首先取当前进程局部描述符表中代码段描述符和数据段描述符项中的段限长（字节数）。
    // 0x0f 是代码段选择符；0x17 是数据段选择符。然后取当前进程代码段和数据段在线性地址
    // 空间中的基地址。由于 Linux 0.12 内核还不支持代码和数据段分立的情况，因此这里需要
    // 检查代码段和数据段基址是否都相同，并且要求数据段的长度至少不小于代码段的长度
    // （参见图 5-12），否则内核显示出错信息，并停止运行。
    // get_limit()和 get_base()定义在 include/linux/sched.h 第 277 行和 279 行处。
44          code_limit=get_limit(0x0f);
45          data_limit=get_limit(0x17);
46          old_code_base = get_base(current->ldt[1]);
47          old_data_base = get_base(current->ldt[2]);
48          if (old_data_base != old_code_base)
49                  panic("We don't support separate I&D");
50          if (data_limit < code_limit)
51                  panic("Bad data_limit");
    // 然后设置创建中的新进程在线性地址空间中的基地址等于（64MB * 其任务号），并用该值
    // 设置新进程局部描述符表中段描述符中的基地址。接着设置新进程的页目录表项和页表项，
    // 即复制当前进程（父进程）的页目录表项和页表项。 此时子进程共享父进程的内存页面。
    // 正常情况下 copy_page_tables()返回 0，否则表示出错，则释放刚申请的页表项。
52          new_data_base = new_code_base = nr * TASK_SIZE;
53          p->start_code = new_code_base;
54          set_base(p->ldt[1],new_code_base);
55          set_base(p->ldt[2],new_data_base);
56          if (copy_page_tables(old_data_base,new_data_base,data_limit)) {
57                  free_page_tables(new_data_base,data_limit);
58                  return -ENOMEM;
59          }
60          return 0;
61  }
62
63  /*
64   *  Ok, this is the main fork-routine. It copies the system process
```

```
65  * information (task[nr]) and sets up the necessary registers.  It
66  * also copies the data segment in it's entirety.
67  */
    /*
     * OK，下面是主要的 fork 子程序。它复制系统进程信息(task[n])
     * 并且设置必要的寄存器。它还整个地复制数据段（也是代码段）。
     */
    // 复制进程。
    // 该函数的参数是进入系统调用中断处理过程（sys_call.s）开始，直到调用本系统调用处理
    // 过程（sys_call.s 第 208 行）和调用本函数前（sys_call.s 第 217 行）逐步压入进程内核
    // 态栈的各寄存器的值。这些在 sys_call.s 程序中逐步压入内核态栈的值（参数）包括:
    // ① CPU 执行中断指令压入的用户栈地址 ss 和 esp、标志 eflags 和返回地址 cs 和 eip;
    // ② 第 83--88 行在刚进入 system_call 时入栈的段寄存器 ds、es、fs 和 edx、ecx、edx;
    // ③ 第 94 行调用 sys_call_table 中 sys_fork 函数时入栈的返回地址（参数 none 表示）;
    // ④ 第 212--216 行在调用 copy_process()之前入栈的 gs、esi、edi、ebp 和 eax（nr）。
    // 其中参数 nr 是调用 find_empty_process()分配的任务数组项号。
68  int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
69                   long ebx,long ecx,long edx, long orig_eax,
70                   long fs,long es,long ds,
71                   long eip,long cs,long eflags,long esp,long ss)
72  {
73          struct task_struct *p;
74          int i;
75          struct file *f;
76
    // 首先为新任务数据结构分配内存。如果内存分配出错，则返回出错码并退出。然后将新任务
    // 结构指针放入任务数组的 nr 项中。其中 nr 为任务号，由前面 find_empty_process()返回。
    // 接着把当前进程任务结构内容复制到刚申请到的内存页面 p 开始处。
77          p = (struct task_struct *) get_free_page();
78          if (!p)
79                  return -EAGAIN;
80          task[nr] = p;
81          *p = *current;  /* NOTE! this doesn't copy the supervisor stack */
                            /* 注意！这样做不会复制超级用户堆栈（只复制进程结构）*/
    // 随后对复制来的进程结构内容进行一些修改，作为新进程的任务结构。先将新进程的状态
    // 置为不可中断等待状态，以防止内核调度其执行。然后设置新进程的进程号 pid，并初始
    // 化进程运行时间片值等于其 priority 值（ 一般为 15 个嘀嗒）。接着复位新进程的信号
    // 位图、报警定时值、会话（session）领导标志 leader、 进程及其子进程在内核和用户
    // 态运行时间统计值，还设置进程开始运行的系统时间 start_time。
82          p->state = TASK_UNINTERRUPTIBLE;
83          p->pid = last_pid;              // 新进程号。也由 find_empty_process()得到。
84          p->counter = p->priority;       // 运行时间片值（嘀嗒数）。
85          p->signal = 0;                  // 信号位图。
86          p->alarm = 0;                   // 报警定时值（嘀嗒数）。
87          p->leader = 0;                  /* process leadership doesn't inherit */
                                            /* 进程的领导权是不能继承的 */
88          p->utime = p->stime = 0;        // 用户态时间和核心态运行时间。
89          p->cutime = p->cstime = 0;      // 子进程用户态和核心态运行时间。
90          p->start_time = jiffies;        // 进程开始运行时间（当前时间滴答数）。
    // 再修改任务状态段 TSS 数据（参见列表后说明）。由于系统给任务结构 p 分配了 1 页新
    // 内存，所以 (PAGE_SIZE + (long) p) 让 esp0 正好指向该页顶端。 ss0:esp0 用作程序
    // 在内核态执行时的栈。另外，在第 3 章中我们已经知道，每个任务在 GDT 表中都有两个
    // 段描述符，一个是任务的 TSS 段描述符，另一个是任务的 LDT 表段描述符。下面 111 行
```

```
    // 语句就是把 GDT 中本任务 LDT 段描述符的选择符保存在本任务的 TSS 段中。当 CPU 执行
    // 切换任务时，会自动从 TSS 中把 LDT 段描述符的选择符加载到 ldtr 寄存器中。
 91         p->tss.back_link = 0;
 92         p->tss.esp0 = PAGE_SIZE + (long) p;  // 任务内核态栈指针。
 93         p->tss.ss0 = 0x10;           // 内核态栈的段选择符（与内核数据段相同）。
 94         p->tss.eip = eip;            // 指令代码指针。
 95         p->tss.eflags = eflags;      // 标志寄存器。
 96         p->tss.eax = 0;              // 这是当 fork()返回时新进程会返回 0 的原因所在。
 97         p->tss.ecx = ecx;
 98         p->tss.edx = edx;
 99         p->tss.ebx = ebx;
100         p->tss.esp = esp;
101         p->tss.ebp = ebp;
102         p->tss.esi = esi;
103         p->tss.edi = edi;
104         p->tss.es = es & 0xffff;     // 段寄存器仅 16 位有效。
105         p->tss.cs = cs & 0xffff;
106         p->tss.ss = ss & 0xffff;
107         p->tss.ds = ds & 0xffff;
108         p->tss.fs = fs & 0xffff;
109         p->tss.gs = gs & 0xffff;
110         p->tss.ldt = _LDT(nr);       // 任务局部表描述符的选择符（LDT 描述符在 GDT 中）。
111         p->tss.trace_bitmap = 0x80000000;       //（高 16 位有效）。
    // 如果当前任务使用了协处理器，就保存其上下文。汇编指令 clts 用于清除控制寄存器 CR0
    // 中的任务已交换（TS）标志。每当发生任务切换，CPU 都会设置该标志。该标志用于管理
    // 数学协处理器：如果该标志置位，那么每个 ESC 指令都会被捕获（异常 7）。如果协处理
    // 器存在标志 MP 也同时置位的话，那么 WAIT 指令也会捕获。因此，如果任务切换发生在一
    // 个 ESC 指令开始执行之后，则协处理器中的内容就可能需要在执行新的 ESC 指令之前保存
    // 起来。捕获处理句柄会保存协处理器的内容并复位 TS 标志。指令 fnsave 用于把协处理器
    // 的所有状态保存到目的操作数指定的内存区域中（tss.i387）。
112         if (last_task_used_math == current)
113                 __asm__("clts ; fnsave %0 ; frstor %0"::"m" (p->tss.i387));

    // 接下来复制进程页表。即在线性地址空间中设置新任务代码段和数据段描述符中的基址
    // 和限长，并复制页表。如果出错（返回值不是 0），则复位任务数组中相应项并释放为
    // 该新任务分配的用于任务结构的内存页。
114         if (copy_mem(nr,p)) {               // 返回不为 0 表示出错。
115                 task[nr] = NULL;
116                 free_page((long) p);
117                 return -EAGAIN;
118         }
    // 如果父进程中有文件是打开的，则将对应文件的打开次数增 1。因为这里创建的子进程
    // 会与父进程共享这些打开的文件。将当前进程（父进程）的 pwd, root 和 executable
    // 引用次数均增 1。与上面同样的道理，子进程也引用了这些 i 节点。
119         for (i=0; i<NR_OPEN;i++)
120                 if (f=p->filp[i])
121                         f->f_count++;
122         if (current->pwd)
123                 current->pwd->i_count++;
124         if (current->root)
125                 current->root->i_count++;
126         if (current->executable)
127                 current->executable->i_count++;
```

```
128         if (current->library)
129                 current->library->i_count++;
    // 随后在 GDT 表中设置新任务 TSS 段和 LDT 段描述符项。这两个段的限长均被设置成 104
    // 字节。参见 include/asm/system.h，52—66 行代码。 然后设置进程之间的关系链表
    // 指针，即把新进程插入到当前进程的子进程链表中。把新进程的父进程设置为当前进程，
    // 把新进程的最新子进程指针 p_cptr 和年轻兄弟进程指针 p_ysptr 置空。接着让新进程
    // 的老兄进程指针 p_osptr 设置等于父进程的最新子进程指针。若当前进程却是还有其他
    // 子进程，则让比邻老兄进程的最年轻进程指针 p_yspter 指向新进程。最后把当前进程
    // 的最新子进程指针指向这个新进程。然后把新进程设置成就绪态。最后返回新进程号。
    // 另外， set_tss_desc() 和 set_ldt_desc() 定义在 include/asm/system.h 文件中。
    // “gdt+(nr<<1)+FIRST_TSS_ENTRY”是任务 nr 的 TSS 描述符项在全局表中的地址。因为
    // 每个任务占用 GDT 表中 2 项，因此上式中要包括'(nr<<1)'。
    // 请注意，在任务切换时，任务寄存器 tr 会由 CPU 自动加载。
130         set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
131         set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
132         p->p_pptr = current;                // 设置新进程的父进程指针。
133         p->p_cptr = 0;                      // 复位新进程的最新子进程指针。
134         p->p_ysptr = 0;                     // 复位新进程的比邻年轻兄弟进程指针。
135         p->p_osptr = current->p_cptr;       // 设置新进程的比邻老兄兄弟进程指针。
136         if (p->p_osptr)                     // 若新进程有老兄兄弟进程，则让其
137                 p->p_osptr->p_ysptr = p;    // 年轻进程兄弟指针指向新进程。
138         current->p_cptr = p;                // 让当前进程最新子进程指针指向新进程。
139         p->state = TASK_RUNNING;        /* do this last, just in case */
140         return last_pid;
141 }
142
    // 为新进程取得不重复的进程号 last_pid。函数返回在任务数组中的任务号(数组项)。
143 int find_empty_process(void)
144 {
145         int i;
146
    // 首先获取新的进程号。如果 last_pid 增 1 后超出进程号的正数表示范围，则重新从 1 开始
    // 使用 pid 号。 然后在任务数组中搜索刚设置的 pid 号是否已经被任何任务使用。如果是则
    // 跳转到函数开始处重新获得一个 pid 号。 接着在任务数组中为新任务寻找一个空闲项，并
    // 返回项号。 last_pid 是一个全局变量，不用返回。如果此时任务数组中 64 个项已经被全
    // 部占用，则返回出错码。
147         repeat:
148                 if ((++last_pid)<0) last_pid=1;
149                 for(i=0 ; i<NR_TASKS ; i++)
150                         if (task[i] && ((task[i]->pid == last_pid) ||
151                                         (task[i]->pgrp == last_pid)))
152                                 goto repeat;
153         for(i=1 ; i<NR_TASKS ; i++)             // 任务 0 项被排除在外。
154                 if (!task[i])
155                         return i;
156         return -EAGAIN;
157 }
158
```