[**fork进程分析**](https://www.cnblogs.com/joey-hua/p/5597818.html)

【版权所有，转载请注明出处。出处：<http://www.cnblogs.com/joey-hua/p/5597818.html> 】

 据说安卓应用里通过fork子进程的方式可以防止应用被杀，大概原理就是子进程被杀会向父进程发送信号什么的，就不深究了。

首先fork()函数它是一个系统调用，在sys.h中：

|  |  |
| --- | --- |
| 1  2  3  4 | extern int sys\_fork ();     // 创建进程。 (kernel/system\_call.s, 208)    // 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。  fn\_ptr sys\_call\_table[] = { sys\_setup, sys\_exit, sys\_fork, ...} |

前面有文章对系统调用做过详细分析，main.c中的：

|  |  |
| --- | --- |
| 1 | static inline \_syscall0 (int, fork) |

将\_\_NR\_fork也就是2和0x80中断绑定了，刚好对应的是上面数组的sys\_fork函数，在system\_call.s中：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #### sys\_fork()调用，用于创建子进程，是system\_call 功能2。原形在include/linux/sys.h 中。  # 首先调用C 函数find\_empty\_process()，取得一个进程号pid。若返回负数则说明目前任务数组  # 已满。然后调用copy\_process()复制进程。  .align 2  \_sys\_fork:  call \_find\_empty\_process        # 调用find\_empty\_process()(kernel/fork.c,135)。  testl %eax,%eax  js 1f  push %gs  pushl %esi  pushl %edi  pushl %ebp  pushl %eax  call \_copy\_process                  # 调用C 函数copy\_process()(kernel/fork.c,68)。  addl $20,%esp                       # 丢弃这里所有压栈内容。  1: ret |

首先调用find\_empty\_process来寻找任务数组中还未使用的编号，在fork.c中：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | // 为新进程取得不重复的进程号last\_pid，并返回在任务数组中的任务号(数组index)。  int find\_empty\_process (void)  {    int i;    repeat:  // 如果last\_pid 增1 后超出其正数表示范围，则重新从1 开始使用pid 号。    if ((++last\_pid) < 0)      last\_pid = 1;    // 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。如果是则重新获得一个pid 号。    for (i = 0; i < NR\_TASKS; i++)      if (task[i] && task[i]->pid == last\_pid)        goto repeat;    // 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。    for (i = 1; i < NR\_TASKS; i++) // 任务0 排除在外。      if (!task[i])        return i;    // 如果任务数组中64 个项已经被全部占用，则返回出处码。    return -EAGAIN;  } |

这个函数比较好理解，接下来看find\_empty\_process的返回值是保存在eax中，如果为负数则直接跳出sys\_fork，否则push一堆指令，作为copy\_process的参数，也在fork.c中：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95 | /\*  \* 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。  \*/  // 复制进程。  // 其中参数nr 是调用find\_empty\_process()分配的任务数组项号。none 是system\_call.s 中调用  // sys\_call\_table 时压入堆栈的返回地址。  int  copy\_process (int nr, long ebp, long edi, long esi, long gs, long none,            long ebx, long ecx, long edx,            long fs, long es, long ds,            long eip, long cs, long eflags, long esp, long ss)  {    struct task\_struct \*p;    int i;    struct file \*f;      p = (struct task\_struct \*) get\_free\_page ();  // 为新任务数据结构分配内存。    if (!p)                                                                       // 如果内存分配出错，则返回出错码并退出。      return -EAGAIN;    task[nr] = p;                                                         // 将新任务结构指针放入任务数组中。                                                                                    // 其中nr 为任务号，由前面find\_empty\_process()返回。    \*p = \*current;        /\* NOTE! this doesn't copy the supervisor stack \*/  /\* 注意！这样做不会复制超级用户的堆栈 \*/ //（只复制当前进程内容）。      p->state = TASK\_UNINTERRUPTIBLE; // 将新进程的状态先置为不可中断等待状态。    p->pid = last\_pid;                                         // 新进程号。由前面调用find\_empty\_process()得到。    p->father = current->pid;                           // 设置父进程号。    p->counter = p->priority;    p->signal = 0;                                                 // 信号位图置0。    p->alarm = 0;                                                  // 报警定时值（滴答数）。    p->leader = 0;     /\* process leadership doesn't inherit \*/  /\* 进程的领导权是不能继承的 \*/    p->utime = p->stime = 0;                                // 初始化用户态时间和核心态时间。    p->cutime = p->cstime = 0;                          // 初始化子进程用户态和核心态时间。    p->start\_time = jiffies;                                   // 当前滴答数时间。  // 以下设置任务状态段TSS 所需的数据（参见列表后说明）。    p->tss.back\_link = 0;    // 由于是给任务结构p 分配了1 页新内存，所以此时esp0 正好指向该页顶端。ss0:esp0 用于作为程序    // 在内核态执行时的堆栈。    p->tss.esp0 = PAGE\_SIZE + (long) p;    // 内核态堆栈指针（由于是给任务结构p 分配了1 页                                                                        // 新内存，所以此时esp0 正好指向该页顶端）。    p->tss.ss0 = 0x10;                                     // 堆栈段选择符（与内核数据段相同）[??]。    p->tss.eip = eip;                                          // 指令代码指针。    p->tss.eflags = eflags;                                // 标志寄存器。    p->tss.eax = 0;                                            // 这是当fork()返回时，新进程会返回0 的原因所在。    p->tss.ecx = ecx;    p->tss.edx = edx;    p->tss.ebx = ebx;    p->tss.esp = esp;                          // 新进程完全复制了父进程的堆栈内容。因此要求task0    p->tss.ebp = ebp;                          // 的堆栈比较“干净”。    p->tss.esi = esi;    p->tss.edi = edi;    p->tss.es = es & 0xffff;                   // 段寄存器仅16 位有效。    p->tss.cs = cs & 0xffff;    p->tss.ss = ss & 0xffff;    p->tss.ds = ds & 0xffff;    p->tss.fs = fs & 0xffff;    p->tss.gs = gs & 0xffff;    p->tss.ldt = \_LDT (nr);                    // 设置新任务的局部描述符表的选择符（LDT 描述符在GDT 中）。    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。如果协处理器存在标志也同时置位的话那么就会捕获    // WAIT 指令。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。错误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）。      if (last\_task\_used\_math == current)      \_\_asm\_\_ ("clts ; fnsave %0"::"m" (p->tss.i387));  // 设置新任务的代码和数据段基址、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内存页。    if (copy\_mem (nr, p))      {                                                       // 返回不为0 表示出错。        task[nr] = NULL;        free\_page ((long) p);        return -EAGAIN;      }  // 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。    for (i = 0; i < NR\_OPEN; i++)      if (f = p->filp[i])        f->f\_count++;  // 将当前进程（父进程）的pwd, root 和executable 引用次数均增1。    if (current->pwd)      current->pwd->i\_count++;    if (current->root)      current->root->i\_count++;    if (current->executable)      current->executable->i\_count++;  // 在GDT 中设置新任务的TSS 和LDT 描述符项，数据从task 结构中取。  // 在任务切换时，任务寄存器tr 由CPU 自动加载。    set\_tss\_desc (gdt + (nr << 1) + FIRST\_TSS\_ENTRY, &(p->tss));    set\_ldt\_desc (gdt + (nr << 1) + FIRST\_LDT\_ENTRY, &(p->ldt));    p->state = TASK\_RUNNING;   /\* do this last, just in case \*/                                                            /\* 最后再将新任务设置成可运行状态，以防万一 \*/    return last\_pid;                      // 返回新进程号（与任务号是不同的）。  } |

这里有问题需要注意一下，为什么copy\_process有那么多参数，而sys\_fork才push了5个寄存器，这是因为根据系统调用机制，调用sys\_fork之前是先调用的system\_call函数，已经往栈压入了一堆寄存器，这就对应上了。

首先**为新任务数据结构分配内存（注意这里是数据结构不是任务本身）**，get\_free\_page放在后面的内存管理文章分析，fork函数和内存管理memory.c是息息相关的。这里只要知道这个函数是获取到主内存区的一页空闲页面并返回这个页面的地址。

接下来的比较好理解，复制当前进程的进程描述符到新任务中，并对各个属性重新赋值。这里值得注意的是p->father = current->pid表示新任务的父进程就是当前进程。

**接下来设置esp0指向刚新分配的页内存的顶端**，ss0为内核数据段选择子，因为内核数据段描述符中的基址为0，所以ss0:esp0用作程序在内核态执行时的堆栈。

接下来p->tss.ldt = \_LDT (nr);设置ldt的索引号，也就是LDT在GDT中的选择子。

下面是最关键的函数copy\_mem：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | // 设置新任务的代码和数据段基址、限长并复制页表。  // nr 为新任务号；p 是新任务数据结构的指针。  int  copy\_mem (int nr, struct task\_struct \*p)  {    unsigned long old\_data\_base, new\_data\_base, data\_limit;    unsigned long old\_code\_base, new\_code\_base, code\_limit;      // 取当前进程局部描述符表中描述符项的段限长（字节数）。    code\_limit = get\_limit (0x0f);    // 取局部描述符表中代码段描述符项中段限长。    data\_limit = get\_limit (0x17);    // 取局部描述符表中数据段描述符项中段限长。    // 取当前进程代码段和数据段在线性地址空间中的基地址。    old\_code\_base = get\_base (current->ldt[1]);    // 取原代码段基址。    old\_data\_base = get\_base (current->ldt[2]);    // 取原数据段基址。    if (old\_data\_base != old\_code\_base)   // 0.11 版不支持代码和数据段分立的情况。      panic ("We don't support separate I&D");    if (data\_limit < code\_limit)   // 如果数据段长度 < 代码段长度也不对。      panic ("Bad data\_limit");    // 创建中新进程在线性地址空间中的基地址等于64MB \* 其任务号。    new\_data\_base = new\_code\_base = nr \* 0x4000000;   // 新基址=任务号\*64Mb(任务大小)。    p->start\_code = new\_code\_base;    // 设置新进程局部描述符表中段描述符中的基地址。    set\_base (p->ldt[1], new\_code\_base);   // 设置代码段描述符中基址域。    set\_base (p->ldt[2], new\_data\_base);   // 设置数据段描述符中基址域。    // 设置新进程的页目录表项和页表项。即把新进程的线性地址内存页对应到实际物理地址内存页面上。    if (copy\_page\_tables (old\_data\_base, new\_data\_base, data\_limit))      {                                                                                       // 复制代码和数据段。        free\_page\_tables (new\_data\_base, data\_limit); // 如果出错则释放申请的内存。        return -ENOMEM;      }    return 0;  } |

首先取局部描述符表（LDT自身的描述符表）中代码和数据段描述符中的限长，在sched.h中：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // 取段选择符segment 的段长值。  // %0 - 存放段长值(字节数)；%1 - 段选择符segment。  #define get\_limit(segment) ({ \  unsigned long \_\_limit; \  \_\_asm\_\_( "lsll %1,%0\n\tincl %0": "=r" (\_\_limit): "r" (segment)); \  \_\_limit;}) |

因为在进程描述符结构中有个

|  |  |
| --- | --- |
| 1 | struct desc\_struct ldt[3];// 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。 |

lsll是加载段界限的指令,即把segment段描述符中的段界限字段装入某个寄存器(这个寄存器与\_\_limit结合),函数返回\_\_limit加1,即段长。

这表示的是LDT描述符表自身，第一个描述符为空，第二个描述符也就是8-15字节是代码段，又因为描述符的0-15位是段限长，是从当前描述符的0位开始，所以取的是0x0f（第二个描述符的0位），然后第三个描述符也就是16-23字节是数据段，所以取0x17.

接下来是取当前进程的ldt的代码段的基地址：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | // 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。  #define \_get\_base(addr) ({\  unsigned long \_\_base; \  \_\_asm\_\_( "movb %3,%%dh\n\t" \   // 取[addr+7]处基址高16 位的高8 位(位31-24)??dh。    "movb %2,%%dl\n\t" \      // 取[addr+4]处基址高16 位的低8 位(位23-16)??dl。    "shll $16,%%edx\n\t" \    // 基地址高16 位移到edx 中高16 位处。    "movw %1,%%dx" \      // 取[addr+2]处基址低16 位(位15-0)??dx。  :"=d" (\_\_base) \        // 从而edx 中含有32 位的段基地址。  :"m" (\*((addr) + 2)), "m" (\*((addr) + 4)), "m" (\*((addr) + 7)));  \_\_base;  }    )  // 取局部描述符表中ldt 所指段描述符中的基地址。  #define get\_base(ldt) \_get\_base( ((char \*)&(ldt)) ) |

current->ldt[1]为当前进程的ldt的代码段描述符项的内容，所以这里就不难理解了，就是从描述符项的内容中提取基地址。

接下来设置新进程的线性地址的基地址，linus**给每个程序（进程）划分了64MB的虚拟内存空间**，所以新基址就是任务号\*64MB。

再接着就是往新进程的LDT表中的段描述符设置基地址了，原理类似。

copy\_page\_tables和free\_page\_tables放到后面一篇讲解。

最后面是设置新任务的TSS和LDT描述符项，在进程调度的初始化中讲解过。

最后返回新进程号。

至此fork函数分析结束。