第三章 进程1的创建

有王伟强《图像处理》和罗平《高级人工智能》这两个课程学得好的同学请联系我！

我这两个课程感觉快挂科了TAT，我操作系统和计算机体系结构学得还行，可以互助一下

email: lujunfeng@junfeng.lu

QQ:464270342

上面是我的联系方式，希望大佬能出手相助

Fork!

```
if (!fork()) {      /* we count on this going ok */
        init();
    }
    //父进程会得到子进程的pid，此处是1
/*
 *   NOTE!!   For any other task 'pause()' would mean we have to get a
 * signal to awaken, but task0 is the sole exception (see 'schedule()')
 * as task 0 gets activated at every idle moment (when no other tasks
 * can run). For task0 'pause()' just means we go check if some other
 * task can run, and if not we return here.
 */
    for(;;) pause();//反复调用pause(),实际上调的是static inline
_syscall0(int,pause)
```

Fork是Linux中用来新建进程的一个函数，如果其返回值就说明当前进程是被fork出来的子进程，反之是父进程。

> 父进程得到的返回值是子进程的pid

可以看出，子进程（进程1）执行了`init()`函数，而父进程（进程0）是在一个死循环中（怠速进程）

Fork函数的生成

Fork实际上是一个系统调用，所以找不到直接的函数定义，因为他是这么展开的

```
#define __NR_setup  0   /* used only by init, to get system going */
#define __NR_exit   1
#define __NR_fork   2
.....
static inline _syscall0(int,fork)
#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_##name)); \
```

```
if (__res >= 0) \
    return (type) __res; \
errno = -__res; \
return -1; \
}
```

```
_sys_fork:
    call _find_empty_process
    testl %eax,%eax
    js 1f
    push %gs        //这个是返回的task[idx]编号
    pushl %esi
    pushl %edi
    pushl %ebp
    pushl %eax
    call _copy_process
    addl $20,%esp //因为这个是两段跳转，所以清一段
1:  ret
```

实际上就是触发了`int 0x80`这个中断。而我们在`sched_init()`里面设置了这样的代码：

```
set_system_gate(0x80,&system_call);
```

即，此时会跑到`_system_call`这个汇编函数去,具体的代码：

```
_system_call:
    cmpl $nr_system_calls-1,%eax
    ja bad_sys_call
    push %ds
    push %es
    push %fs
    pushl %edx
    pushl %ecx      # push %ebx,%ecx,%edx as parameters
    pushl %ebx      # to the system call
    movl $0x10,%edx     # set up ds,es to kernel space
    mov %dx,%ds
    mov %dx,%es
    movl $0x17,%edx     # fs points to local data space
    mov %dx,%fs
    call _sys_call_table(,%eax,4)   //调用中断表
    pushl %eax  //把返回值压栈
    movl _current,%eax  //当前进程
    ……
```

注意看这个`_sys_call_table`,他定义在`sys.h`中：

```
fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, ……
```

即,最后跳到了`_sys_fork`这个函数当中去

```
_sys_fork:
    call _find_empty_process
    testl %eax,%eax
    js 1f
    push %gs
    pushl %esi
    pushl %edi
    pushl %ebp
    pushl %eax
    call _copy_process
    addl $20,%esp //因为这个是两段跳转，所以清一段
1:  ret
```

这个fork函数最终是长这样的。他调用了两个函数：

- find_empty_process
- copy_process

Fork第一步，找到空进程

```
int find_empty_process(void)
{
    int i;

    repeat:
        if ((++last_pid)<0) last_pid=1;
        for(i=0 ; i<NR_TASKS ; i++)
            if (task[i] && task[i]->pid == last_pid) goto repeat;
    for(i=1 ; i<NR_TASKS ; i++)
        if (!task[i])
            return i;
    return -EAGAIN;
}
```

其中，这里有两个概念

第一个是`last_pid`，这个是一个全局计数器，表明这个进程是系统创建以来的第几个进程。这个数字只增不减(溢出除外)。

> 其实这个值叫做new_pid可能会好点，因为是用来给下一个进程用的

第二个是`task_idx`即，进程控制块的下标，表明这个进程用的是`task[]`数组中的第几个控制块。

第一个循环是找到一个合法的`last_pid`，首先这个值不能为负（因为这个数会一直加，万一溢出就成负数了），其次，这个值不能和正在运行的进程是一样的pid，比如溢出后，从1从头开始算了，但是进程1还在运行中，此时就不能把pid给新的进程

第二个循环，就是返回一个空的进程控制块（编号）了。

Fork第二步，拷贝数据

相关的函数如下：

```
                    //nr:之前找到的空进程块
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
                 //这个none，是systemcall调用sys_fork的时候，压进去的返回地址
        long ebx,long ecx,long edx,
        long fs,long es,long ds,//这两行是syscall压进去的

        long eip,long cs,long eflags,long esp,long ss)//这行是int 0x80自动压进去的
{
    ……
}
```

他的参数，有一部分是在汇编中压进去的。有一部分是在进行函数调用/中断的时候系统自动压进去的。

申请新的页

为了拷贝数据，必须要有一个地方存放数据，因此此时需要新申请一页的空间。

```
    struct task_struct *p;
    int i;
    struct file *f;

    p = (struct task_struct *) get_free_page();
    if (!p)
        return -EAGAIN;
```

这个get_free_page函数，在这里引用一下：

```
//// 取空闲页面。如果已经没有可用内存了，则返回 0。
 // 输入：%1(ax=0) - 0；%2(LOW_MEM)；%3(cx=PAGING PAGES)；
%4(edi=mem_map+PAGING_PAGES-1)。
 // 输出：返回%0(ax=页面起始地址)。
 // 上面%4 寄存器实际指向 mem_map[]内存字节图的最后一个字节。本函数从字节图末端开始向前扫描
 // 所有页面标志（页面总数为 PAGING_PAGES），若有页面空闲（其内存映像字节为 0）则返回页面地
址。
 // 注意！本函数只是指出在主内存区的一页空闲页面，但并没有映射到某个进程的线性地址去。后面
 // 的 put_page()函数就是用来作映射的。
63 unsigned long get_free_page(void)
64 {
65 register unsigned long __res asm("ax");
66
67 __asm__("std ; repne ; scasb\n\t" // 方向位置位，将 al(0)与对应每个页面的(di)内容比
较，
68 "jne 1f\n\t" // 如果没有等于 0 的字节，则跳转结束（返回 0）。
69 "movb $1,1(%%edi)\n\t // 将对应页面的内存映像位置 1。
70 "sall $12,%%ecx\n\t" // 页面数*4K = 相对页面起始地址。
71 "addl %2,%%ecx\n\t" // 再加上低端内存地址，即获得页面实际物理起始地址。
72 "movl %%ecx,%%edx\n\t" // 将页面实际起始地址îedx 寄存器。
73 "movl $1024,%%ecx\n\t" // 寄存器 ecx 置计数值 1024。
74 "leal 4092(%%edx),%%edi\n\t" // 将 4092+edx 的位置îedi(该页面的末端)。
75 "rep ; stosl\n\t" // 将 edi 所指内存清零（反方向，也即将该页面清零）。
76 "movl %%edx,%%eax\n" // 将页面起始地址îeax（返回值）。
77 "1:"
78 :"=a" (__res)
79 :"" (0),"i" (LOW_MEM),"c" (PAGING_PAGES),
```

```
80 "D" (mem_map+PAGING_PAGES-1)
81 :"di","cx","dx");
82 return __res; // 返回空闲页面地址（如果无空闲也则返回 0）。
83 }
```

std：set td，设置方向位置为1，从高到低。

repne scasb： 扫描edi指向的字符串，如果遇到字节等于al或者ecx计数为0，则结束扫描

实际上是在扫描这个数组,看有没有没用过的页，找到后，把该页清零

```
static unsigned char mem_map [ PAGING_PAGES ] = {0,};
```

制作子进程的进程控制块

```
struct task_struct *p;
.....
task[nr] = p;
*p = *current;  /* NOTE! this doesn't copy the supervisor stack */
p->state = TASK_UNINTERRUPTIBLE;
p->pid = last_pid;
p->father = current->pid;
p->counter = p->priority;
......
p->tss.eip = eip;//这里的eip是控制了fork之后，子进程从哪执行的
......
p->tss.eax = 0;//注意：这里设置了返回值为0
......
p->tss.ldt = _LDT(nr);
```

```
#define FIRST_TSS_ENTRY 4
#define FIRST_LDT_ENTRY (FIRST_TSS_ENTRY+1)
#define _LDT(n) ((((unsigned long) n)<<4)+(FIRST_LDT_ENTRY<<3))
```

注意，由于`*p`是一个`task_struct`而不是一个`task_union`，所以拷贝的时候并没有把后面的栈部分也给拷贝

由于是Fork，大部分东西"看起来"和父进程一样，所以一开始是直接复制父进程的`task_struct`，但是后面还需要修改一些东西.

这里需要特别注意`p->tss.eax = 0;`这一句话。

tss实际上就是中断还原的现场，在这里把`eax`即返回值设置成了0，这也就是为什么`fork()`之后，子进程得到的返回值为`0`

同时`p->tss.eip = eip;`这一句话，使得父子进程返回的地方是一样的（调用了syscall的地方）

然后每一个进程都有自己的LDT（局部描述符表），用来指定这个进程运行时的代码段和数据段。

这里的`_LDT(n)`实际上是一个宏定义，是寻找GDT上的对应项

给子进程在虚拟地址（线性地址）上分配页表

```
if (copy_mem(nr,p)) {
        task[nr] = NULL;
        free_page((long) p);
        return -EAGAIN;
    }
```

```
#define get_limit(segment) ({ \
unsigned long __limit; \
__asm__("lsll %1,%0\n\tincl %0":"=r" (__limit):"r" (segment)); \
__limit;})
```

lsll 是加载段界限的指令，把 segment 段描述符中的段界限字段装入`__limit`,然后再加1

即，得到段限长

```
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;

    code_limit=get_limit(0x0f);//01111,即内核代码段的段限长
    data_limit=get_limit(0x17);//10111，内核
    old_code_base = get_base(current->ldt[1]);//
    old_data_base = get_base(current->ldt[2]);
    if (old_data_base != old_code_base)
        panic("We don't support separate I&D");
    if (data_limit < code_limit)
        panic("Bad data_limit");
    new_data_base = new_code_base = nr * 0x4000000;//设置每一个进程的目录基址，是进程
数*64MB
    p->start_code = new_code_base;
    set_base(p->ldt[1],new_code_base);
    set_base(p->ldt[2],new_data_base);

    if (copy_page_tables(old_data_base,new_data_base,data_limit)) {
        free_page_tables(new_data_base,data_limit);
        return -ENOMEM;
    }
    return 0;
}
```

在这一版的Linux中，每一个进程只能用64MB的虚拟空间，这些进程分割总共4G的虚拟空间。

一个进程可用的空间为`[nr*64MB,nr*64MB+64MB]`

在最开始，子进程的地址空间和父进程的地址空间是一模一样的，为了使得父子进程的内存空间分开，需要把相应的目录项给复制到新的地址

```
|----|++++|----|----|----|----|----|----|
|----|++++|----|++++|----|----|----|----|
```

复制前后的地址示意，上面`-`号代表未在页目录中分配的虚拟地址。

具体的拷贝函数如下：

```
int copy_page_tables(unsigned long from,unsigned long to,long size)
{
    unsigned long * from_page_table;
    unsigned long * to_page_table;尽
    unsigned long this_page;
    unsigned long * from_dir, * to_dir;
    unsigned long nr;

    if ((from&0x3fffff) || (to&0x3fffff))   //地址不对齐就报错
        panic("copy_page_tables called with wrong alignment");
    // 0xffc=111111111100,把4以下的清空
    from_dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */ //取地址
的MB数，然后再算地址的4MB数，刚好是一个页目录项管理的4MB大小（1024页表项*4K=4M）
    to_dir = (unsigned long *) ((to>>20) & 0xffc);
//也就是，取页目录项
    size = ((unsigned) (size+0x3fffff)) >> 22;  //size / 4MB     向上取整
    for( ; size-->0 ; from_dir++,to_dir++) {
        if (1 & *to_dir)
            panic("copy_page_tables: already exist");
        if (!(1 & *from_dir))   //不拷贝空项
            continue;
        from_page_table = (unsigned long *) (0xfffff000 & *from_dir);
        if (!(to_page_table = (unsigned long *) get_free_page()))
            return -1;  /* Out of memory, see freeing */
        *to_dir = ((unsigned long) to_page_table) | 7;
        nr = (from==0)?0xA0:1024;       // 因为内核的就用了160项。(ldt算一下)
        for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {//子循环，复制目录项
            this_page = *from_page_table;
            if (!(1 & this_page))   //不拷贝空项
                continue;
            this_page &= ~2;    //禁止写原有的项，为了触发写时复制机制
            *to_page_table = this_page;
            if (this_page > LOW_MEM) {//LOWMEM是1M以下的内核区域，不参加分页。
                *from_page_table = this_page;   //共享的地方，谁都别写，谁写谁去触发写
时复制
                this_page -= LOW_MEM;   //从1M才开始管理页表
                this_page >>= 12;
                mem_map[this_page]++;
            }
        }
    }
    invalidate();//刷新缓存
    return 0;烦
}
```

简单来说就是，先拷贝页目录再拷贝页表项，同时，不拷贝空项。

拷贝完成之后，所有的项都不能写，谁写谁触发写时复制机制

设置文件系统相关，以及tss和ldt

```
for (i=0; i<NR_OPEN;i++)
        if (f=p->filp[i])
            f->f_count++;
    if (current->pwd)
        current->pwd->i_count++;
    if (current->root)
        current->root->i_count++;
    if (current->executable)
        current->executable->i_count++;
    set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
    set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
```

这一段，上方是文件系统相关的东西，主要就是把打开的文件引用计数增加1，表示多了一个正在共享文件的进程

然后下方是设置这个子进程的tss段和ldt段。

这样这个进程就准备好独立运行了

（父进程）Fork函数的收尾&返回

```
p->state = TASK_RUNNING;    /* do this last, just in case */
return last_pid;
```

一切就绪，此时把进程的状态设置为可运行，接着返回。

_sys_fork

此时运行的还是父进程，因此返回到了`_sys_fork`函数中

```
_sys_fork:
    call _find_empty_process
    ....
    pushl %esi
    pushl %edi
    .....
    call _copy_process
    addl $20,%esp //父进程返回到这，然后清上面push进去的参数，接着返回
1:  ret
```

这个`_sys_fork`函数是`_system_call`调用的，因此会返回到`_system_call`中

_system_call

```
_system_call:
    .....
    call _sys_call_table(,%eax,4)    #之前 在这调用的 sys_fork
    pushl %eax  #把返回值压栈 即，子进程的pid
    movl _current,%eax
    cmpl $0,state(%eax)     # state 查看当前进程状态是否为0 /* -1 unrunnable, 0
runnable, >0 stopped */
    jne reschedule
    cmpl $0,counter(%eax)       # counter 这个是用来计时间片的,如果当前进程不运行，或者
剩余时间片没了，那么重新调度
    je reschedule
```

在这，先保存了返回的子进程`pid`值，然后看一下当前进程是否可运行，如果不可运行，那直接跳去调度函数中，切到别的进程，否则的话继续运行下面的代码：

```
ret_from_sys_call:  //否则正常返回
    movl _current,%eax      # task[0] cannot have signals
    cmpl _task,%eax     //103 行上的_task 对应 C 程序中的 task[]数组，直接引用 task 相当于引用 task[0]。
    je 3f
    。。。。
3:  popl %eax   //之前把eax给压栈了，现在把返回值给拿回来。
    popl %ebx
    popl %ecx
    popl %edx
    pop %fs
    pop %es
    pop %ds
    iret
```

判断了一下当前进程是否为进程0，如果是进程0，那么直接返回。

注意，由于我们是进系统调用了，因此这个返回是用`iret`返回

fork()

我们进入系统调用，是在fork()函数里面，之前说过了，这是一个宏定义生成的函数

```
#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_##name)); \
if (__res >= 0) \
    return (type) __res; \
errno = -__res; \
return -1; \
}
```

因此我们实际上是返回到了`if(__res >= 0)`这一行代码中。

如果子进程的pid是大于等于0的，说明这个fork运行正常了，那么返回子进程pid，否则返回-1表示异常，并且记录`errno`

main()

```
.....
if (!fork()) {      /* we count on this going ok */
        init();
    }
for(;;) pause();//反复调用pause(),实际上调的是static inline _syscall0(int,pause)
```

由于父进程此时会拿到子进程的pid（非0），因此if会失效，到下面的死循环中。

进程0进入idle状态

```
int sys_pause(void)
{
    current->state = TASK_INTERRUPTIBLE;//设置当前进程进入可中断的等待状态
    schedule();
    return 0;
}
```

进程1实际上就是反复处于一个死循环中，把自身设置为可中断的等待状态，然后放弃自身运行，试图调度其他的进程。

（子进程）Fork函数的返回

子进程当前处于可运行的状态，但是最开始运行的是进程0，而不是进程1.

那么什么时候会切换到进程1去运行呢?

- 父进程调用了`schedule()`函数
- 或者时钟中断来了，进行了进程的切换

最终都是要来到同一个函数

schedule()进程调度函数

```
void schedule(void)
{
    int i,next,c;
    struct task_struct ** p;

/* check alarm, wake up any interruptible tasks that have got a signal */

    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p) // 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
        if (*p) {
            if ((*p)->alarm && (*p)->alarm < jiffies) {//r如果设置过了alarm,并且现在的时间已经过了alarm的时间
                    (*p)->signal |= (1<<(SIGALRM-1));//然后发送信号
                    (*p)->alarm = 0;//清空alarm
                }
            if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
            (*p)->state==TASK_INTERRUPTIBLE)
                (*p)->state=TASK_RUNNING;
        }

/* this is the scheduler proper: */

    while (1) {
        c = -1;
        next = 0;
        i = NR_TASKS;//#define NR_TASKS 64
        p = &task[NR_TASKS];
        while (--i) {
            if (!*--p)
                continue;   //跳过空进程
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
        }//挑选最大counter的进程，并且下标是i,因为至少有一个进程有效，所以至少c是0（没有时间片了）
```

```
        if (c) break;//要么还有counter，要么所有进程都挂了（-1），所有进程都挂了的情况，那么会调度到进程0
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)//如果都找不到大于0的counter，且可运行的进程，那么把所有的进程counter更新一下
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) +// counter = oldcounter/2 + priority，这样子，在等待中的进程恢复运行后，会有更多的时间片
                        (*p)->priority;
    }
    switch_to(next);
}
```

这个函数分为两个部分。

- 第一个部分是检查是否有需要定时唤醒的进程，如果已经到了需要唤醒的时间，那么就把这些进程设置为可运行
- 第二个部分则是试图找到一个进程满足：
    - 可运行
    - 还有空余时间片

    然后试图调度这个进程。

    如果出现例外情况：

    - 没有可运行的进程
        - 强制调度进程0
    - 可运行的进程时间片都用完了
        - 更新时间片

switch_to(n)切换进程

```
#define switch_to(n) {\
struct {long a,b;} __tmp; \ /*定义一个临时的数据结构*/
__asm__("cmpl %%ecx,_current\n\t" \ /*ecx=task[n] 如果task[n]和 current是同一个东西，那就不调度了*/
    "je 1f\n\t" \
    "movw %%dx,%1\n\t" \                //把_tss移动到task.b
    "xchgl %%ecx,_current\n\t" \        //交换， current = task[n]；ecx = 被切换出的任务
    "ljmp %0\n\t" \                                                 // 执行长跳转至__tmp.a,__tmp.b ，a是偏移，b是tss，即 ljmp 0,_tss(n)
                                                                    // 在任务切换回来后才会继续执行下面的语句。
    "cmpl %%ecx,_last_task_used_math\n\t" \
    "jne 1f\n\t" \
    "clts\n" \      // 新任务上次使用过协处理器，则清 cr0 的 TS 标志。
    "1:" \
    ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
    "d" (_TSS(n)),"c" ((long) task[n])); \
}
```

真正进行进程的切换，是靠这个函数完成的。

这用了一种很奇怪的进程切换方式，`ljmp`

他是试图用一个结构体进行跳转

```
struct{
    long offset;
    long tss;
}
```

即，通过tss切换进程，还原之前进程的现场。

fork ()

此时，通过`ljmp`指令，进程已经切换到了进程1.

这种切换方式会自动地让进程从`eip`处开始运行，而这eip是之前父进程在调用`int 0x80`即系统调用时所记录的地址。

即，和父进程一样（只是跳过了中间syscall那几层返回），来到了fork()函数中

```
#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_##name)); \
if (__res >= 0) \
    return (type) __res; \
errno = -__res; \
return -1; \
}
```

由于之前在设置进程控制块中，设置了`p->tss.eax = 0;`

也就是让子进程认为自己得到的返回值是0。因此，此时会把`__res=0`给再返回main函数

main ()

因此，到了main函数这，就会进到if内部，从而进到init()函数了。至此，进程1正式运行。

```
if (!fork()) {      /* we count on this going ok */
    init();
}
```