**深入分析fork的执行过程**

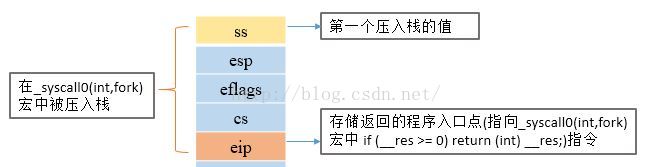
2016年08月09日 00:54:19 [u010132427](https://me.csdn.net/u010132427) 阅读数：1333

        在上一篇文章中简单分析了fork、pause等系统调用的实现，怀着对fork在父子进程中返回不同值的好奇，本文中将深入分析fork的执行过程以及如何实现在父子进程中返回不一样的值(父进程---子进程ID，子进程----0)。

        为了分析fork，可以从它定义处开始一步一步的分析它执行的过程以及堆栈内容的变化。下面从syscall0(int,fork)展开后的结果：

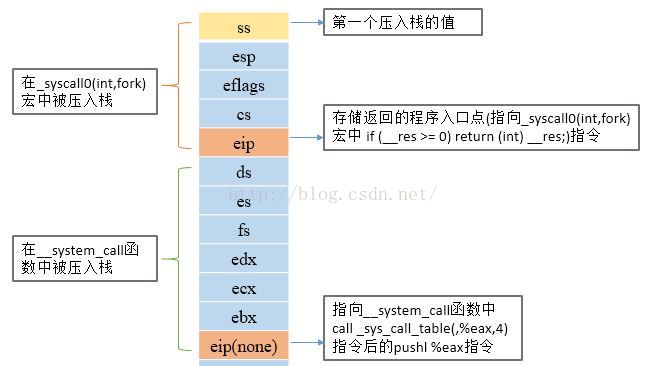
1. static inline int fork(void)
2. {
3. long \_\_res;
4. \_\_asm\_\_ volatile ("int $0x80" \ *//调用系统中断0x80*
5. : "=a" (\_\_res) \ *//\_\_res用来承载中断返回值*
6. : "0" (\_\_NR\_fork)); \ *//输入为系统中断调用号\_\_NR\_fork ( = 2)*
7. if (\_\_res >= 0) \
8. return (int) \_\_res; \ *//如果返回值>=0，则直接返回该值。*
9. errno = -\_\_res; \ *//否则置出错号*
10. return -1; \ *//并返回-1*
11. }

       从上面第2行代码可知，fork执行过程的起点为“int $0x80” ，通过调用系统中断0x80从而跳转到\_system\_call中去执行。返回值\_\_res从eax寄存器中得到，当\_\_res >= 0时返回\_\_res值，否则报错并返回-1 。在调用系统中断0x80时，[CPU](https://www.baidu.com/s?wd=CPU&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)保存现场，自动把一些寄存器的值按顺序压入栈，此时栈内的内容如下图所示，把ss、esp、eflags、cs寄存器入栈，在调用system\_call函数时把返回程序的入口地址也入栈。



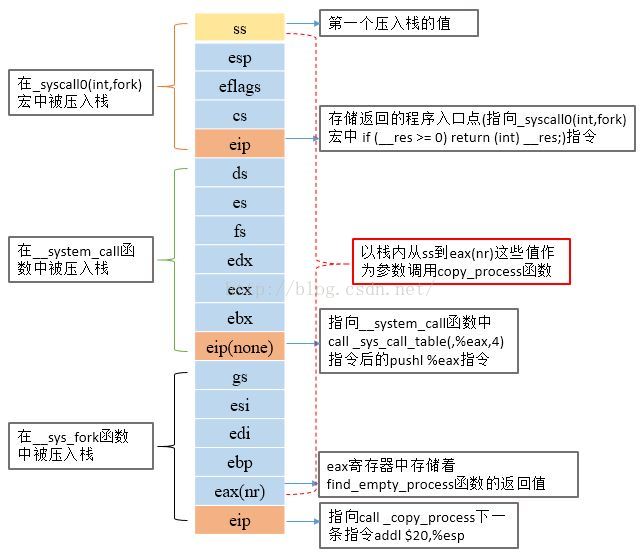
         接下来，程序跳转进入/kernel/system\_call.S文件中system\_call系统调用入口函数\_system\_call处执行。

1. .align 2
2. reschedule:
3. pushl $ret\_from\_sys\_call
4. jmp \_schedule
5. .align 2
6. \_system\_call:
7. cmpl $nr\_system\_calls-1,%eax
8. ja bad\_sys\_call
9. push %ds
10. push %es
11. push %fs
12. pushl %edx
13. pushl %ecx # push %ebx,%ecx,%edx as parameters
14. pushl %ebx # to the system call
15. movl $0x10,%edx # set up ds,es to kernel space
16. mov %dx,%ds
17. mov %dx,%es
18. movl $0x17,%edx # fs points to local data space
19. mov %dx,%fs
20. call \_sys\_call\_table(,%eax,4)
21. pushl %eax *//%eax刚好是\_sys\_fork:中call \_copy\_process 的返回值----last\_pid，即 子进程号*
22. movl \_current,%eax
23. cmpl $0,state(%eax) # state
24. jne reschedule
25. cmpl $0,counter(%eax) # counter
26. je reschedule
27. ret\_from\_sys\_call:
28. movl \_current,%eax # task[0] cannot have signals
29. cmpl \_task,%eax
30. je 3f
31. cmpw $0x0f,CS(%esp) # was old code segment supervisor ?
32. jne 3f
33. cmpw $0x17,OLDSS(%esp) # was stack segment = 0x17 ?
34. jne 3f
35. movl signal(%eax),%ebx
36. movl blocked(%eax),%ecx
37. notl %ecx
38. andl %ebx,%ecx
39. bsfl %ecx,%ecx
40. je 3f
41. btrl %ecx,%ebx
42. movl %ebx,signal(%eax)
43. incl %ecx
44. pushl %ecx
45. call \_do\_signal
46. popl %eax
47. 3: popl %eax
48. popl %ebx
49. popl %ecx
50. popl %edx
51. pop %fs
52. pop %es
53. pop %ds
54. iret

       在\_\_system\_call中，先把ds,es,fs,edx,ecx,ebx压入栈，再把call \_sys\_call\_table(,%eax,4)指令的返回入口地址入栈，然后调转到sys\_fork函数处执行，此时栈内的内容为：  


1. .align 2
2. \_sys\_fork:
3. call \_find\_empty\_process
4. testl %eax,%eax
5. js 1f
6. push %gs
7. pushl %esi
8. pushl %edi
9. pushl %ebp
10. pushl %eax
11. call \_copy\_process
12. addl $20,%esp
13. 1: ret

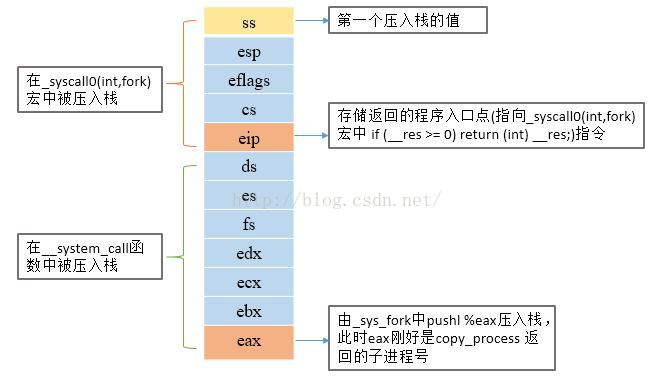
      在sys\_fork中，首先调用find\_empty\_process函数取得不重复的进程号，其返回值在eax寄存器中；然后把gs,esi,edi,ebp,eax寄存器的值压栈，然后，以前面压入栈内的ss到eax(nr)这些值为参数调用copy\_process函数，此时栈内的内容为：



      在copy\_process函数中有如下两行代码对于fork生成的子进程很关键：

1. p->tss.eip = eip; *//新的进程b的TSS里头的eip指向 syscall0中的 （if (\_\_res >= 0) return (int) \_\_res;）指令*
2. ...
3. p->tss.eax = 0; *//新的进程b的TSS里头的eax赋值为0，当调度新进程运行时，新进程的syscall0中返回值\_\_res = p->tss.eax = 0（即在新进程中fork返回的进程号为0）*

      执行完copy\_process函数后，返回addl $20,%esp指令处，把栈顶指针esp上移20(栈内的5个项\* 4 字节 = 20)，刚好把gs,esi,edi,ebp,eax(nr)的空间忽略掉。然后使用ret指令返回eip(none)中的地址，即\_\_system\_call函数中的pushl %eax指令处，把eax寄存器的值(为copy\_process 返回的子进程号)入栈。此时栈内的内容变为：



  此时，程序返回到\_system\_call中，接下来判断是否进行进程调度：

1. movl \_current,%eax
2. cmpl $0,state(%eax) # state
3. jne reschedule
4. cmpl $0,counter(%eax) # counter
5. je reschedule

  这些代码的意思是：先比较当前current，即子进程的状态是否可以运行，如果当前进程不再就绪状态就去执行调度程序，如果该任务在就绪状态但时间片用完了，也就执行调度程序。所有后续的情况是，我们无法确定进程0或者进程1先执行，但是返回值已经明显确定了。

a. 对于进程0(父进程)而言，接下来会执行ret\_from\_sys\_call后的指令，进行system\_call系统调用的退出和信号处理，其中call \_do\_signal后面的popl %eax 表示把 子进程号 出栈存到eax中，返回到syscall0时传递给\_\_res，表示进程0(父进程)的fork返回的子进程号。  
b. 对于进程1(子进程)而言，在schedule函数中调度到子进程运行时，由前面copy\_process函数可知，子进程会返回到syscall0中if(\_\_res >= 0) return (int) \_\_res;)指令处，\_\_res为 0，即子进程的fork返回0 ，其过程如下：

1. .align 2
2. reschedule:
3. pushl $ret\_from\_sys\_call
4. jmp \_schedule

  先把\_ret\_from\_sys\_call的地址压入栈，再跳转到schedule函数执行，在schedule函数最后会调用switch\_to宏：

1. #define switch\_to(n) {\
2. struct {long a,b;} \_\_tmp; \
3. \_\_asm\_\_("cmpl %%ecx,\_current\n\t" \ *// 比较当前任务current和要切换到的任务task[n]*
4. "je 1f\n\t" \ *// 如果要切换到的任务是当前任务，则跳到标号1，即结束，什么也不做，否则继续执行下面的代码*
5. "movw %%dx,%1\n\t" \ *// 把新任务的TSS选择符\_TSS(n) 赋值给 \_\_tmp.b的低16位*
6. "xchgl %%ecx,\_current\n\t" \ *// 交换两个操作数的值，相当于C代码的：current = task[n] ，ecx = 被切换出去的任务（原任务）；*
7. "ljmp %0\n\t" \ *// 长跳转到地址&\_\_tmp.a中包含的48bit逻辑地址处：\_\_tmp.a即为该逻辑地址的offset部分，*
8. *// \_\_tmp.b的低16bit为seg\_selector(高16bit无用)部分， 即切换到选择符\_TSS(n)指定的的任务*
10. "cmpl %%ecx,\_last\_task\_used\_math\n\t" \ *// 返回原进程后开始执行指令的地方。*
11. "jne 1f\n\t" \
12. "clts\n" \
13. "1:" \ *// 返回\_ret\_from\_sys\_call处*
14. ::"m" (\*&\_\_tmp.a),"m" (\*&\_\_tmp.b), \
15. "d" (\_TSS(n)),"c" ((long) task[n])); \
16. }

  switch\_to宏的核心是"ljmp %0\n\t"指令，它实现任务的切换。当它切换到子进程时，由于子进程的tss.eip指向syscall0中if(\_\_res >= 0) return (int) \_\_res;)指令处，且tss.eax = 0，所以子进程中fork会返回0值。